TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Liam Simonos Warren 223695IVCM

Analysis of CDOC 2.0 Protocols in ProVerif

Master's Thesis

Supervisor: Nikita Snetkov
MSc

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Liam Simonos Warren 223695IVCM

CDOC 2.0 protokollide analüüs ProVerifis

Magistritöö

Juhendaja: Nikita Snetkov
MSc

Tallinn 2024

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Liam Simonos Warren

14/04/2024

Abstract

The 2017 Estonian ID card crisis revealed vulnerabilities contained in Estonian ID cards and lead to the development of the CDOC 2.0 protocol. This thesis examines ECDH and RSA communication schemes outlined in CDOC 2.0 and proves confidentiality within these protocols. The proof of confidentiality is achieved through analysis with the cryptographic protocol verifier ProVerif. ProVerif uses symbolic reasoning to create proofs of confidentiality for protocols with cryptographic primitives. These primitives are outlined in detail within the paper, and a thorough account of their abstraction is given. Findings from ProVerif suggest that both of the examined protocols are effective in protecting the confidentiality of some secret data. This thesis aims to increase confidence in the security of CDOC 2.0 protocols and introduce the importance of automatic proof verification.

This thesis is written in English and is 49 pages long.

List of abbreviations and terms

DH	Diffie-Hellman
ECDH	Elliptic curve Diffie-Hellman
RSA	Rivest-Shamir-Adleman
XOR	Exclusive OR (logic)
HKDF	HMAC key derivation function

Table of contents

1 Introduction

1.1 Motivation

In 2017, the Estonian ID card crisis brought to light the need for a new approach for data encryption via ID cards [1]. Previously, it was found that the prime numbers generated for key pairs in the old ID cards could be brute forced due to a small seed space [14]. A new protocol, CDOC 2.0, has been proposed as a more secure means of transmitting and storing encrypted data [1]. Because the focus of this implementation is to protect data, namely confidentiality, it is therefore important to verify that the new approach is in fact secure. This thesis has made use of a cryptographic protocol verifier called ProVerif to demonstrate the security of select CDOC 2.0 communication schemes. The benefit this study provides is an assurance that the protocols used within CDOC 2.0 are logically proven to maintain confidentiality.

1.2 Research Questions

- What does an analysis of CDOC 2.0 protocols in ProVerif state about their security?
    - How is ProVerif able to accomplish this analysis?
    - What are the horn clauses used in these protocols?
    - Which aspects of security related to CDOC 2.0 can be verified with this tool?

1.3 Scope and Goal

The primary outcome of this research has been to verify the confidentiality of two protocols within CDOC 2.0. The scope of analysis has been limited to the ECDH [20] and RSA [21] communication protocols within CDOC 2.0, and ProVerif is the tool used to perform this analysis. In addition to a proof of the security of protocols within CDOC

2.0, an overview of the essential horn clauses within these proofs has been provided in section 5. ProVerif itself has some assumptions behind it; it treats cryptographic primitives as black boxes. This means that it is assumed that a primitive, like RSA, is secure, and ProVerif does not account for the details of these primitives. This study therefore also assumes that the primitives in these protocols are secure and only analyzes the protocols themselves.

The outcome of this analysis of protocols has been the validation that the confidentiality of the protocols is maintained. If the work of this thesis revealed an attack on these protocols, there would be an ethical responsibility to disclose the method of attack to the responsible party. However, no such attack has been found. The protocols have been shown to maintain confidentiality of information, but in the case of an attack, attack traces outlined by ProVerif [9] detail precisely how the attack can be performed. These traces can be used to find a solution for resolving a vulnerability within a protocol.

1.4 Novelty

As will be shwn in section 2, numerous protocols have already been examined via ProVerif. The primary novelty of this study is the selection of protocols. The protocols examined by this thesis have not previously been publicly verified in ProVerif. The contribution is the verification of confidentiality in CDOC 2.0's protocols, and this differs from other experiments with ProVerif because the targeted protocols are not the same. The approach has been similar to other works because the protocols have been translated into a form that is usable by ProVerif. But, because the protocols are different, a new combination of principles has been tested.

1.5 Preliminaries

This section will give a brief overview of fundamental and cryptographic principles that are used in the protocols examined in this thesis. Diffie-Hellman (DH) [19] key exchange is the first important primitive for this paper. Because the specific mathematics of a given type of DH is not required for ProVerif due to its level of abstraction, this overview does not elaborate on the various mathematical operations used in different DH schemes. The purpose of DH key exchange is to provide two parties with a symmetric key over an unsecure channel [13]. Given parties A and B, both parties have a private value and access to some shared generator G. Party A takes their private value, $privA$ and combines it with G to produce a public value $pubA$. Party B does a similar process with their private value, and both parties exchange $pubA$ and $pubB$ with one another over a public channel.

One important property of these public values is the fact that the private values of either party cannot be feasibly derived from the public value. Another important property of the operation is the fact that this process of combination—symbolized by * for example—will produce an equivalent final value such that $(privA * G) * pubB = (privB * G) * pubA$. The result of this combination is the symmetric key. Because of the two properties just outlined, parties A and B will have the same symmetric key through the exchange of a value over a public network, and no other party can create the symmetric from the public information.

The second main cryptographic primitive used in the examined protocols is RSA, or because it will be abstracted, public key cryptography. Public key cryptography provides a method for encrypting messages for a specific recipient that only that recipient can decrypt [13]. This is an asymmetric form of cryptography which means unlike DH, there is not a shared key. Any participant in a public cryptography scheme has a private and public key pair. These two keys are "paired" because they have a special relationship where information encrypted by a public key can be decrypted by the paired private key. The public key can also be applied to a message encrypted with the parties' private key to authenticate the sender, and this is the basis for signatures. The logical relationship for encryption and decryption is the following: given a message m, $decrypt(privKeyA, encrypt(pubKeyA, m)) = m$. This relationship shows that encryption takes a public key and a message, and decrypting with the corresponding private key will produce that message.

XOR is another foundational operation used in the protocols examined. Typically, XOR, denoted by $\oplus$, is a logical operator that outputs the value 1 if only one of two inputs is 1. That is to say, in a binary table with $0 \oplus 0, 1 \oplus 0$,$0 \oplus 1, 1 \oplus 1$, only $1 \oplus 0$ and $0 \oplus 1$ output 1. This operation can be applied to a bitstring of some length, and a consequence of its construction is the fact that given bitstrings x and y, $x \oplus (x \oplus y) = y \, and \, y \oplus (x \oplus y) = x$. This logical relationship is utilized in the CDOC protocols as a means of encryption. The reduction of the above primitives introduces the importance of symbolic modelling [2] in ProVerif. By treating these primitives abstractly, it is easier to reason about the processes on a logical level.

2 Literature Review

The main focus of this review is to explore how ProVerif is applied to protocols, to establish a knowledge basis for protocols used in CDOC 2.0, and it will also introduce the research gap relative to ProVerif. *Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif* provides a solid introduction into the uses and capabilities of ProVerif [2]. Blanchet lists many uses for verifying security protocols such as "e-commerce, wireless networks, credit cards, [and] e-voting" [2]. This formal method of verifying protocols provides assurances that functional testing cannot [2]. Blanchet introduces two types of models, symbolic and computational, and proposes that symbolic models are better suited for automatic verification tools [2]. Symbolic models treat cryptographic primitives as black boxes [2] which means these models are not directly concerned with how the primitives function. Computational models, however, focus on low level processes [2]. This simplification of using black boxes compounded with a more generalized perspective is what makes symbolic models more suited for automatic verification.

In the context of CDOC 2.0 protocols, it is assumed that cryptographic primitives, such as RSA, are secure. Any vulnerabilities will arise from the improper combination of primitives or the transference of secret information over a public channel. The basic functions of a primitive, like the interaction between encryption and decryption, must still be modelled in order to give an accurate representation of their security properties. In a similar manner to the public key cryptography in the previous section, the reduction of asymmetric encryption can be expressed as: $reduc\ forall\ m{:}\ bitstring\ , k{:}\ skey;\ adec(aenc(m, pk(k))\ , k)\ =\ m$ [2]. In this example, there is a message m which is encrypted by a constructor aenc which takes the message m, and a public key pk(k) related to a secret key k. The decryption constructor adec takes the result of the encryption and a secret key k as input, and it is described that for any message m and secret key k, if a message is encrypted using those variables, the decryption will output the original message m. Here it is shown how functions like asymmetric encryption can remain abstract while still capturing the essence of these protocols.

The way that ProVerif ultimately achieves a representation of protocols is through the use of Horn clauses [2]. Horn clauses maintain relational information with messages [2], which seems to mean that the context and connection between messages is maintained. ProVerif translates protocols into the horn clauses which are used to prove security properties. The derivability of various facts is tested, and if the facts are not derivable, then security is proven [2]. This implies that if a fact is derivable, then some adversary is able to deduce information about it. Proving security depends on the inability to deduce information from some fact. ProVerif is able to verify secrecy and authentication [2] which is one its main features.

Queries are necessary for finding vulnerabilities in protocols using ProVerif. Queries can either be made from the attacker's perspective through the query of some value x [2], or events can be placed throughout the protocol to perform more advanced queries called "correspondences" [2]. Events can serve multiple purposes; an event can be placed within a process to show if that step of the process has been reached, or multiple events can be queried together to see the order in which events occur. This query on the sequence of events, called a "correspondence assertion" [2] can be used to confirm that an event e happens only after another event e' has already occurred. This can be helpful for confirming that a party is not performing an encryption or decryption process before receiving the required keys. The validity of events can be further examined with an "injective correspondence" which not only checks the order of events but also ensures a one to one correspondence between events. This means that if a process is run multiple times, event e only occurs one time for every e' that occurs.

ProVerif is capable of examining a number of protocols; here are some examples. The paper gives an example of a key exchange protocol called "Denning-Saco" [2]. The provided example mostly serves as a model for how to describe a protocol within ProVerif, but it does not give proof of security. This model shows the encryption, decryption, and signature verification process of keys and a secret *s* [2]. It is important that the information here is available to an adversary. Secrecy is proved through the assumption that an adversary can access the transmission of messages [2]. This makes sense because the focus of the protocol should be that intercepted messages cannot be decrypted, not that messages cannot be intercepted. Other protocols like resistance to Denial of service attacks [3], 5G TLS handshakes [4], and ZRTP [5] have all been examined by ProVerif. These protocols have already been examined by ProVerif, but the gap in the research lies in the protocols that have not been analyzed.

One potential issue that this paper introduces is the representation of XOR processes [2]. It is stated that XOR cannot be expressed with constructors, but at the end of the paper, there is the suggestion that horn logic can overcome this problem [2]. The paper *Reducing Protocol Analysis with XOR to the XOR-free Case in the Horn Theory Based Approach* provides a solution to this issue. In general, it demonstrates how XOR can be reduced to what the authors call the "XOR-free case" [6]. Because ProVerif cannot deal with XOR, this work around is a significant contribution considering that many cryptographic operations rely on XOR. The reduction itself happens through a process called "syntactic derivation" [6]. The theorem of the paper is stated as a message can only be derived from T if it can be derived from T+ [6]. Here, T and T+ represent the horn model before and after applying reduction [6].

The methodology of the paper largely relies on experimentation. It provides practical examples of their reduction approach by using ProVerif, and they are able to evaluate the efficiency of their method. Furthermore, they were able to discover a new attack vector that had not previously been found [6]. One other paper, *On the Automatic Analysis of Recursive Security Protocols with XOR*, also examines the connection between XOR and horn logic. This paper posits that the commonly perceived undecidability of XOR can actually be decided in recursive XOR protocols [7]. Here, decidability refers to whether or not some security property is valid or invalid within the system, and recursive protocols are protocols where repetitive actions take place.

The paper takes a somewhat similar approach to the previous by transforming the XOR problem into a problem without XOR [7]. One difference is its focus is on a particular class of protocols with recursive functions. Both papers about XOR demonstrate the importance of solving a XOR problem in the context horn logic. Furthermore, these papers give an indication at the end that the methods described within may help with Diffie-Hellman exponentiation [7]. Through these suggestions, a further research gap is revealed.

Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation attempts to take on this problem of DH exponentiation by creating a syntactical derivation problem [8]. This syntactical derivation avoids algebraic issues related to DH. The authors show that this derivation can be applied to a class of problems called "exponent-ground horn theories" [8]. If the terms belonging to a clause only contain subterms without variables, then the clause can be called grounded [8]. In the conclusion, the authors state that the focus of the research has been on secrecy, but that it may be possible to branch into other fields of security concerns [8].

For the purposes of the exploration of this paper, the modelling of DH provided in the ProVerif manual will suffice for capturing DH processes in the CDOC 2.0 protocols. The equation "equation forall x: exponent , y: exponent; exp(exp(g,x),y) = exp(exp(g,y),x)" [9] is given, and here essential properties of DH are presented. Namely, given a generator g and private exponents x and y, the order of exponentiation—whether x is exponentiated with x or y first—is unimportant and will yield the same result. It should be stated that, like many processes in ProVerif, exponentiation here is abstracted and any actual mathematical exponentiating does not occur. This equation simply describes the relationship between two variables of an abstract type exponent and a constant g of an abstract type generator. The value of the constructor exp(g,x) and exp(g,y) when nested in the constructor exp() with y and g respectively produce an equal value, and this shows equivalence as it is expressed in Diffie-Hellman exponentation.

In addition to ProVerif, there are a number of tools which also are used for automated proofs. Some of these tools are Tamarin, CryptoVerif, and EasyCrypt. Like ProVerif, Tamarin is a symbol modelling tool [16]. Tamarin includes an interactive mode which allows users to examine security proofs in greater detail [16]. While Tamarin allows for advanced modelling by creating different states in a protocol, its complexity leaves ProVerif as a more suitable program for the scope of this thesis. CryptoVerif takes a different approach from ProVerif and uses a computational model rather than a symbolic model [17]. The advantages of symbolic models were discussed earlier in this section, and a symbolic model seemed most appropriate for the analysis of the target protocols. In comparison to ProVerif, EasyCrypt requires a higher level of interaction [18]. The construction of proofs requires activity from the user [18] compared to ProVerif's automated approach. EasyCrypt's interactive and game-based approach captures protocol nuances effectively, but it could overly broaden the project's scope.

3 Research Methods

In general, the methodology of this paper is a research-based approach to proving the security of a set of protocols. More specifically, ProVerif as a tool is a method for achieving these proofs of security. Given the assumptions of secure cryptographic primitives in conjunction with the axioms used in ProVerif, this tool can logically prove that a given protocol is either secure or unsecure. The foundations of the tool are built on pi-calculus and horn clauses which are means of reasoning about processes used in security protocols. The application of these fields branches further into theoretical computer science and programming logic, but in the context of this thesis, they aid in examining protocols.

ProVerif takes an input pi-calculus file that describes a cryptographic protocol. Pi-calculus is significant because it allows for handling concurrent processes [9]. Cryptographic protocols can be defined as a way of communicating information over some channel with the goal of keeping that information secure. Both channels and the information that an attacker has access to are defined within ProVerif [9]. For names, such as RSA, within the pi-calculus file, queries are made against them to determine if the attacker can derive them or not. The query will return true if it is not derivable, and false if it is derivable [9]. ProVerif provides an "attack trace" [9] which shows the method the attacker used to derive a name if such a derivation occurred. The analysis of these traces can reveal weaknesses within a protocol.

ProVerif follows the Dolev-Yao model [10] which means that the attacker has significant control within the environment; it can "read, modify, delete, and inject messages" [9] within the communication channels. The attacker can manipulate data, but it cannot perform cryptographic operations unless it has the required keys. Once a pi-calculus file is run in ProVerif, the output contains equations, processes, queries, goal, attack derivation, attack trace, and the query result [9]. This thesis has focused on analysis of the goal, attack derivation, attack trace, and query result because they directly pertain to the security of the tested protocols. In the case of confidentiality, the goal is whether or not the examined property has remained secret. The attack derivation and trace show the exact method the attacker used to learn the secret if it was able to, and the query result summarizes if goal has been achieved [9]. The attack derivation is presented in English while the attack trace is given in pi calculus.

The methodology has been to examine these attack derivations and traces to understand which properties have been shown to be secure or unsecure. In the case that a property is unsecure, a list of steps is given in the derivation that shows, for example, how the authentication of party B to A has been broken. The derivations of these properties have served as the basis for examining the security of the protocols in this paper, and the validation is contained within the analysis of ProVerif itself.

ProVerif's manual [9] provides numerous examples and studies that can be used understand its processes and queries. To demonstrate the process of the methodology, it will be helpful to analyze a simple handshake protocol.

```
let clientA(pkA:pkey,skA:skey,pkB:spkey) =
        out(c,pkA);
        in(c,x:bitstring);
        let y = adec(x,skA) in
        let (=pkB,k:key) = checksign(y,pkB) in
        event acceptsClient(k);
        out(c,senc(s,k));
        event termClient(k,pkA).

let serverB(pkB:spkey,skB:sskey,pkA:pkey) =
        in(c,pkX:pkey);
        new k:key;
        event acceptsServer(k,pkX);
        out(c,aenc(sign((pkB,k),skB),pkX));
        in(c,x:bitstring);
        let z = sdec(x,k) in
        if pkX = pkA then event termServer(k).

process
        new skA:skey;
        new skB:sskey;
        let pkA = pk(skA) in out(c,pkA);
        let pkB = spk(skB) in out(c,pkB);
        ( (!clientA(pkA,skA,pkB)) | (!serverB(pkB,skB,pkA)) ) [9]
```

The full script for this protocol is available in appendix 2. This protocol describes the communication between a client and a server represented by clientA and serverB. These two processes are run in parallel in the main process "process." The main constructors used in this script are symmetric encryption, asymmetric encryption, and digital signatures. The client and server perform an exchange where the client sends its public key pkA, and the server receives it as pkX and creates a symmetric key k. After creating the key k, the server encrypts and signs a tuple of its public key pkB and symmetric key k. pkX is used for encryption so that party A can decrypt it, and it is signed with skB to provide authentication that the server is the signer. A receives the message from the server, decrypts it with their secret key skA, and checks the signature using B's public key skB. Now that A has the symmetric key, it can encrypt a secret s with k and send it to B. B receives the ciphertext x, and decrypts it to create z which should be equal to the original secret s.

```
free s:bitstring [private].
query attacker(s).

event acceptsClient(key).
event acceptsServer(key,pkey).
event termClient(key,pkey).
event termServer(key).

query x:key,y:pkey; event(termClient(x,y))==>event(acceptsServer(x,y)).
query x:key; inj-event(termServer(x))==>inj-event(acceptsClient(x)).
```

This seemingly standard protocol can be analyzed through the use of events and queries. Query attacker(s) is the first query that is declared, and it examines the secrecy of message s. As previously mentioned, attacker queries show if the attacker can derive a secret within the protocol. The four events which pertain to the server and client are used to check correspondence between the actions of the server and client. The query x:key,y:pkey; event(termClient(x,y))==>event(acceptsServer(x,y)) checks if the end of the process from the client only happens after the acceptance from the server, and the value of the public key y is compared to ensure the value has not changed. Query x:key; inj-event(termServer(x))==>inj-event(acceptsClient(x)) examines a similar property but it examines if the end of the server process only follows after the end of the client acceptance. The "inj" in this query is the injective correspondence query, so it checks for one to one correspondence between the events. It will now be helpful to analyze the query output of this script.

```
Verification summary:
RESULT not attacker(s []) is false .
RESULT event(termClient(x 2 , y 1)) ==> event(acceptsServer(x 2 ,y 1)) is false .
RESULT inj−event(termServer(x 2)) ==> inj−event(acceptsClient(x 2)) is true .
```

There are two problems presented in the output. The first result states that the attacker has been able to derive the secret s, so the secrecy of the message in the protocol is broken. The second result shows there is some problem with the correspondence. Either the order of the events is incorrect, the keys x are not equal, or there is a problem with the public keys y. Because the third result does not have any problems, and it is unlikely that the terminal event would be executed before the acceptance event, there is probably an issue with the public keys. Fortunately, ProVerif has provided a trace for the attack on secret s. The trace is presented as text in the output, but it can also be output graphically. The attack trace for this protocol can be found in appendix 3.

In the attack trace, a series of vertical lines can be found which represent the clientA, serverB, and attacker from left to right. The horizontal arrows represent communication between parties. For example, the first two messages exchanged are the public keys of A and B over the public channel, and these are available to the attacker because the attack has access to the public channel. Problems start to occur when the attack sends its own public key pk(a_1) to the server, and there is an event acceptsServer(k_3,pk(a_1)) where the server accepts the attackers public key. The server then sends a signed and encrypted message to the attacker which the attacker can decrypt and re-encrypt using its public key. Using the message received from the server, the attacker sends the newly encrypted message to clientA and impersonates the sever. The client responds and sends the attacker an encrypted message, and the attacker is able to decrypt it by using the information it obtained from the server.

Preventing this attack only requires a small change to the protocol. For clientA, “let (=pkB,k:key) = checksign(y,pkB) in” is changed to “let (=pkA,=pkB,k:key) = checksign(y,pkB) in”. In serverB, “out(c,aenc(sign((pkB,k),skB),pkX));” is changed to “out(c,aenc(sign((pkX,pkB,k),skB),pkX));” The primary difference between the protocols is that A confirms that the provided public key comes from party B which is expressed by “=B”. This is possible because now B includes its public key in the triple that it signs which is shown by the addition of pkB within sign(). With these two changes, partyA no longer accepts communication from the attacker, and the output of the protocol is:

```
RESULT not attacker(s []) is true .
RESULT event(termClient(x 2 , y 1)) ==> event(acceptsServer(x 2 ,y 1)) is true .
RESULT inj−event(termServer(x 2)) ==> inj−event(acceptsClient(x 2)) is true .
```

This example has demonstrated some of the basic properties that can be examined by ProVerif and how the output of queries can explain attack vectors. The examination of the protocols for this thesis follows a similar methodology to determine secrecy for communication between two parties.

4 Results

4.1 Direct Communication ECDH Scheme

To discuss the results of the ProVerif Analysis of CDOC 2.0 protocols, it is first necessary to elaborate the features of the protocols examined in this paper. The first of two protocols is direct communication with ECDH [11]. The protocol is as follows [11]:

```
$1. A: FMK \leftarrow GenerateKeyExtract(Nonce)$
$2. A: CEK \leftarrow GenerateKeyExpand(FMK)$
$3. A: C \leftarrow Encrypt(CEK, M)$
$4. A\ gathers\ public\ keys\ of\ all\ recipients: PK1, PK2, ..., PK\ell;$
  $corresponding\ secret\ keys\ are\ SK1, SK2, ..., SK\ell$
$5. A: (KEK\_i, Capsule\_i) \leftarrow Encapsulate(PKi)$
$6. A: CK\_i \leftarrow XOR(KEK\_i, FMK)$
$7. A\ sends\ the\ following\ to\ each\ recipient\ Bi\ (i\ from\ 1\ to\ \ell):$
  $-\ Encrypted\ message\ C$
  $-\ Encrypted\ key\ CK\_i$
  $-\ Key\ capsule\ Capsule\_i$
$8. Bi: KEK\_i \leftarrow Decapsulate(Capsule\_i, SKi)$
$9. Bi: FMK \leftarrow XOR(KEK\_i, CK\_i)$
$10. Bi: CEK \leftarrow GenerateKeyExpand(FMK)$
$11. Bi: M \leftarrow Decrypt(CEK, C)$
```

In general, this protocol describes the transmission of a secret message M to from party A to some number of B parties who receive and decrypt the message. The protocol begins by extracting a value fmk from a nonce and then that value fmk is expanded to create the value cek. cek is used to encrypt the message m resulting in ciphertext c. A then takes the public key[s] of the recipient[s] and performs encapsulation; this creates a symmetric encryption key kek*i* and a capsule which contains A's ephemeral public key that is generated through ECDH. Party A XORs kek*i* with fmk to produce ck*i*, and transmits c,ck*i*, and caps*i* over a channel to the corresponding B party.

B receives the information over the channel and performs decapsulation with the capsule—A's ephemeral public key—and their private key to produce the symmetric key kek*i* through ECDH. Because in steps 6 and 9 the kek*i* values are equivalent, B can reproduce the value fmk through XORing kek*i* and ck*i*. With fmk, B can now expand it to create cek. Because cek was used to encrypt message M, B now decrypts cipher c and retrieves the original message.

Before presenting the ProVerif script for this protocol, it is necessary to outline some changes between the protocol above and the ProVerif representation. Two primary changes have been made to limit the scope of this thesis: first, the resulting script features Diffie-Hellman but not elliptic curve Diffie-Hellman, and second, the script does not include communication with multiple B parties. The reason that elliptic curve is ignored is because it provides a detailed explanation of how the public key is obtained, but the abstract representation of an exchanged public key that is computed with a user's private key remains the same. While it may be possible to model ECDH in ProVerif, it would broaden the scope of this paper without providing proportional insights into the security of the selected protocol.

For a similar reason, the property of communication with multiple B parties has also been omitted. If A is able to securely send a message to party B_1, then sending a message to party B_2 using the same protocol should not introduce any additional challenges because there is no communication between the two B parties. The attacker already has the opportunity to attempt to insert their own key information, so it is as if the attacker is trying to impersonate a valid B party. Including multiple parties in the ProVerif script would be possible, but once again, it introduces unnecessary complexity without revealing anything about the security of the protocol.

A full version of the final script can be found in appendix 4. For the first part of the script, in addition to type bitstring, there are four types that are used in this protocol. There are types to represent public keys, parties, exponents, and generators. Public keys are used to represent the public key of B needed for DH, and parties are used to establish identity between groups A and B. Exponents and generators are used in DH constructors where G is the result of exponentiation such as the ephemeral public key and final DH key, and exponents are the private values for each party. The queries and events will be discussed in detail in the analysis section of this paper. B's exponent, expB, is taken as a private constant because it is only available to B, and the public key of B will not change even with multiple occurrences of the protocol.

```
(* HKDF functions *)
fun hkdf_extract(bitstring): bitstring.
fun hkdf_expand(bitstring): bitstring.

(* DH *)
const expB: exponent [private].
const g: G.
fun exp(G, exponent): G.
equation forall x: exponent, y: exponent; exp(exp(g, x), y) = exp(exp(g, y), x).

(* XOR function, [12]*)
fun xor(bitstring,bitstring):bitstring.
equation forall x:bitstring, y:bitstring; xor(xor(x,y),y)=x.
equation forall x:bitstring; xor(x,xor(x,x))=x.
equation forall x:bitstring; xor(xor(x,x),x)=x.
equation forall x:bitstring, y:bitstring; xor(y,xor(x,x))=y.
equation forall x:bitstring, y:bitstring; xor(xor(x,y),xor(x,x))= xor(x,y).
equation forall x:bitstring, y:bitstring; xor(xor(x,y),xor(y,y))= xor(x,y).

(* Symmetric encryption and decryption *)
fun senc(bitstring, bitstring): bitstring.
reduc forall m:bitstring, k:bitstring; sdec(senc(m,k),k) = m.

(* Type conversion *)
fun gToBitstring(G) : bitstring [data, typeConverter].
fun gToPkey(G) : pkey [data, typeConverter].
reduc forall g:G; pkeyToG(gToPkey(g)) = g.

(* secret b/t two parties *)
fun m(party,party) : bitstring [private]. (* private because attacker cannot derive secret just from party
names *)
```

This script includes the following constructors/functions: HKDF, DH, XOR, and symmetric key encryption. There are also functions to convert between types. This type conversion is needed for XOR and public key storage. XOR takes type bitstring as its input, and because dhKey is used in the XOR process, it must be converted to a bitstring. For converting between G and pkey, a destructor is used to guarantee the equality of the converted values. The HKDF functions do not have a destructor which describes their logical interaction, so they are included to provide a more accurate representation of the given ECDH protocol. The DH functions, as discussed in section 2, include an equation to express that $(g^\wedge x)^\wedge y = (g^\wedge y)^\wedge x$ are indeed equal. This essentially expresses that the order of exponentiation of private values x and y are unimportant. To reiterate, the constructor exp() is symbolic and does not actually compute exponentiation, so this same algebraic relationship can be said to be true for ECDH if one imagines exp() to represent multiplication of points on an elliptic curve.

The XOR constructor and equations cover a multitude of possible XOR calculations, the most fundamental of which are the first two equations. Symmetric encryption and decryption, as the name suggests, requires a destructor that allows a message to be both encrypted and decrypted by the same shared key k. One final important constructor is “fun m(party,party) : bitstring [private].” This constructor takes two party names, parties A and B in the case of this protocol and produces a private bitstring. Here, the bitstring represents the secret message A wants to send to B, and it is private because the message cannot be retrieved only by knowing the names of the parties A and B.

```
let honestUser(A: party, B: party) =

        event honest(A);
        (processA(A, B))
        |
        (processB(B, A)).

let dishonestUser =
        new name: party;
        in(c, (expX : exponent));
        let dhX = exp(g, expX) in
        let pkX = gToPkey(dhX) in

        insert pkeys(name,pkX);
        out (c, (name,pkX));

        0.

(* Main process *)
process
        new partyA: party;
        new partyB: party;
        let dhB = exp(g, expB) in
        let pkB = gToPkey(dhB) in
        insert pkeys(partyB, pkB);

        (!honestUser(partyA, partyB) | !dishonestUser)
```

To understand the user processes, it will be helpful to work top down from the main process. From “!honestUser(partyA, partyB) | !dishonestUser”, one can see that there are two processes ‘honestUser’ and ‘dishonestUser.’ The “|” symbol means these are run in parallel, and the “!” symbol means that they are run numerous times. The process dishonestUser is simulating an unwanted party that is trying to find a private exponent over the channel c and create their own public key from that exponent. One other important part of the script set out at the beginning is something called a table declared by “table pkeys(party,pkey).” As described by the manual, a table is used for persistent storage that cannot be accessed by the attacker [9]. In the case of this script, the table is used to store the public key of party B. The two parties A and B are declared in this process, and the public key of B is generated and placed into the table. Another key is placed in the table in the process dishonestUser. The existence of the dishonest party ensures that a private exponent has not been leaked and an honest party cannot be impersonated. If the dishonest party were able to retrieve a private exponent, then they would be able to create a public key of an honest party, store it in the table with their name, and begin communication with the opposite party. Process honestUser is primarily used for the event honest(user) which will be explained more in the analysis.

```
let processA (A: party, B : party) =

        (* Generating CEK from FMK using HKDF *)
        new nonce:bitstring;
        let fmk = hkdf_extract(nonce) in
        let cek = hkdf_expand(fmk) in
        event viewBeginA(A,B, cek);

        (* Message encryption *)
        let cipher = senc(m(A,B), cek) in *)

        (* DH encapsulation *)
        new expA : exponent;
        let gWithA = exp(g, expA) in

        (* Retrieves PK of B *)
        get pkeys(=B, pkB: pkey) in

        (* DH Completion *)
        let dhB = pkeyToG(pkB) in
        let gWithAandB = exp(dhB, expA) in (* encapsulation *)
        let dhKey = gToBitstring(gWithAandB) in

        (* A XORs *)
        let ckB = xor(fmk, dhKey) in

        (* Sends info to B *)
        out(c, (cipher, ckB, gWithA));

        event aFinished();
        event viewEndA(A,B, cek);
        0.

let processB (B : party, A : party) =

        (* Receive info from A *)
        in (c, (cipher1:bitstring, ckB1: bitstring, gWithA: G));

        (* Decapsulation *)
        let gWithBandA = exp(gWithA, expB) in

        (* DH completion *)
        let dhKey = gToBitstring(gWithBandA) in

        (* XOR and cek derivation *)
        let fmk1 = xor(ckB1, dhKey) in
        let cek1 = hkdf_expand(fmk1) in
```

```
event viewBeginB(A,B, cek1);

(* Decryption *)
let (=m(A,B)) = sdec(cipher1, cek1) in
event viewEndB(A,B, cek1);
0.
```

processA begins by creating a bitstring “nonce” that will be used for the creation of cek. The first two constructors create the value fmk from nonce and then cek from fmk—CDOC steps 1 and 2. Note that the value fmk is required to create cek, and fmk will be retrieved by party B to create an equal cek value for decryption. The cipher is then created with cek and a message m (step 3). Here one can observe that constructor m(A,B) is used as the value for the message. A and B are party names provided into the input of processA, and this constructor attaches the identity of the parties to the message itself. This association will be important for queries made within the script.

In order for party B to retrieve fmk and create the symmetric key cek for decryption, the two parties must first establish a symmetric DH key. From the outline of the steps provided by CDOC 2.0, it is not explicit how DH takes place during this protocol. This script interprets the protocol to express that encapsulation (step 4) and decapsulation (step 8) are the action of sending an ephemeral public key and creating the final symmetric key respectively. What is labelled as kek*i* in CDOC is the dhKey in the script because this is the symmetric key established through Diffie-Hellman. As an aside, the creation of kek is outlined in section 6.3.4.1 of CDOC 2.0 [11], but because the extraction and expansion functions do not have a describable destructor and the generation of kek is not repeated by B, the establishment of the symmetric key dhKey is taken as a sufficient expression of the protocol for the purposes of this script.

Returning to processA, one can see that A creates its ephemeral public key gWithA through the constructor exp(g, expA) where g is a constant and expA is the private value for A. To complete the DH key, a retrieves B’s public key from the key table. In the “get pkeys” line, =B provides the input of the party name for party B to retrieve the correct key. If processA used =A instead, for example, the process could not complete because A has not entered a key into the public key table. A then exponentiates B’s public key with their private exponent to create the completed DH key. Type conversion is needed here for exponentiation and then again to present the DH key as a bitstring for the upcoming XOR operation.

Party A should not send fmk directly over the channel because an attacker could use hkdf_expand(fmk) to reconstruct cek, so A performs a XOR operation with fmk and dhkey to create ckB. The encrypted message, ckB, and A's public ephemeral key are all sent over the channel to B. B receives the information and exponentiates A's ephemeral public key with B's private exponent to create the DH key on their side. Because B has a symmetric dhKey, B is able to XOR the ckB1 that they received with their DH key to recover fmk1. It is a property of XOR that if XOR(a,b) → c then XOR(a,c) → b where b is the value for fmk in this instance. Normally, XOR should be associative, but because only some properties of XOR have been modelled in the ProVerif script, changing let ckB = xor(fmk, dhKey) to let ckB = xor(dhKey, fmk) will yield a different result. This demonstrates some of the complexity of modelling XOR in ProVerif. Now that B has fmk1, B is able to create cek1 through hkdf_expand(fmk1). Because cek is the encryption key, B can now decrypt the message and receive m(A,B) which concludes the protocol.

4.2 Direct Communication RSA Scheme

The following is the protocol for the RSA communication scheme:

1. $A : fmk \leftarrow GenKeyExtractSym(Nonce)$
2. $A: cek \leftarrow GenKeyExpandSym(FMK)$
3. $A: c \leftarrow Enc(CEK, M)$
4. $A: keki \leftarrow GenKeySym\ (i = 1,2,\ldots,\ell)$
5. $A: cki \leftarrow XOR(keki, fmk)\ (i = 1,2,\ldots,\ell)$
6. $A\ gathers\ public\ keys\ of\ all\ recipients: PK1, PK2, \ldots, PK\ell;$
   $corresponding\ secret\ keys\ are\ SK1, SK2, \ldots, SK\ell$
7. $A : capsi \leftarrow EncryptRSA(PKi, keki)\ (i = 1,2,\ldots,\ell)$
8. $A\ sends\ the\ following\ to\ each\ recipient\ Bi\ (i\ from\ 1\ to\ \ell):$
   $-\ Encrypted\ message\ C$
   $-\ Encrypted\ key\ CK\_i$
   $-\ Key\ capsule\ Capsule\_i$
9. $Bi : keki \leftarrow DecryptRSA(SKi, capsi)$
10. $Bi : fmk \leftarrow XOR(keki, cki)$
11. $Bi : cek \leftarrow GenKeyExpandSym(fmk)$
12. $Bi : M \leftarrow Decrypt(cek, c)$

There are many similarities between this scheme and the ECDH scheme. The main differences are the lack of DH key exchange and the presence of public key cryptography. In step 7, the capsule caps$i$ is created through RSA encryption which takes the recipients public key and a generated value kek$i$. In step 9, party B decrypts caps$i$ by using the private key which corresponds to the public key used for encryption, and kek*i* is retrieved which allows for the recovery of fmk. The ECDH scheme created kek*i* through the encapsulation process, but here there is a new process GenKeySym that creates the value.

Like the ProVerif script for ECDH, multiple B parties are not taken into account because it does not affect the confidentiality of a secret between two parties. The creation of kek*i* is handled slightly different between the ECDH and RSA scripts. In ECDH, kek*i* was treated as equivalent to the created DH key for the reasons previously discussed. In the RSA script, however, kek*i* is declared as a private bitstring outside of the processes for a couple of reasons. First, because there is only one B party, multiple kek values are not needed, so a single value is sufficient. Second, the modelling of a process GenKeySym would not contribute anything to the analysis of the confidentiality of the script because there is no reverse process. For these reasons, kek*i* is declared independent of process A as can be seen with:

```
free kekB : bitstring [private].
```

The modelling of the public key cryptography is:

```
fun pk(skey): pkey.
fun aenc(pkey, bitstring): bitstring.
reduc forall m:bitstring, k:skey; adec(k, aenc(pk(k),m)) = m.
```

The constructor pk takes a secret key as input and produces a public key from that secret key. This is used to create a key pair which ensures there is a connection between the secret and public keys. aenc and adec—asymmetric encryption and decryption—take a public key and private key respectively along with a bitstring to encrypt or decrypt a bitstring. The reduction describes that for any bitstring m, the decryption of a message encrypted with that m will produce the same m. There is the additional condition that the public key used to encrypt the message must be derived from the private key used to decrypt the message; this is shown through pk(k). The main process makes use of this pk() constructor by creating a key pair for B that is inserted into a public key table.

```
(* Main process *)
process
        new partyA: party;
        new partyB: party;
        let pkB = pk(skB) in (* Creates key pair for B *)
        insert pkeys(partyB, pkB);

        (!honestUser(partyA, partyB) | !dishonestUser)
```

Unlike the ECDH scheme were the public key was created through the use of a private exponent, here the public key is constructed with pk().

Processes A and B will look mostly familiar with the main difference being the use of public key cryptography.

```
let processA (A: party, B : party) =

        (* Generating CEK from FMK using HKDF *)
        new nonce:bitstring;
        let fmk = hkdf_extract(nonce) in
        let cek = hkdf_expand(fmk) in
        event viewBeginA(A,B, cek);

        (* Message encryption *)
        let cipher = senc(m(A,B), cek) in

        (* A XORs *)
        let ckB = xor(fmk, kekB) in

        (* Retrieves PK of B *)
        get pkeys(=B, pkB: pkey) in

        (* PK encapsulation *)
        let capsB = aenc(pkB, kekB) in

        (* Sends info to B *)
        out(c, (cipher, ckB, capsB));

        event aFinished();
        event viewEndA(A,B, cek);
        0.

let processB (B : party, A : party) =

        (* Receive info from A *)
        in (c, (cipher1:bitstring, ckB1: bitstring, capsB1: bitstring));

        (* Decapsulation *)
        let kekB1 = adec(skB, capsB1) in

        (* XOR and cek derivation *)
        let fmk1 = xor(ckB1, kekB1) in
        let cek1 = hkdf_expand(fmk1) in
        event viewBeginB(A,B, cek1);

        (* Decryption *)
        let (=m(A,B)) = sdec(cipher1, cek1) in (* checks if the decryption gives original message
*)  event bFinished();
        event viewEndB(A,B, cek1);
        0.
```

The encapsulation step in processA uses constructor aenc with parameters pkB and kekB. pkB is B’s public key placed in the public key table in the main process, and it is retrieved by A in the preceding line. As was discussed, kekB is declared outside of process A, and these two values form caps B. In processB, when party B decapsulates, it uses its secret key and the capsB1 sent by A to create kekB1. kekB1 allows the retrieval of fmk1 through XORing, and this then leads to cek1. With cek1, the message m(A,B) can be decrypted resulting in the completion of processB.

5 Analysis

5.1 Direct Communication ECDH and RSA analysis

For the ECDH and RSA scripts, the events, queries, and verification outputs are all the same, and the following analysis applies to both scripts. There are seven events and five queries present within the ECDH and RSA scripts. The first two events, aFinished() and bFinished(), can be thought of as debugging events. These are markers used to see if that point of the script is reachable or not. aFinished() is executed near the end of processA to show whether or not everything in that process has successfully executed. The same is true for processB, and it holds special significance because of the preceding line “let (=m(A,B)) = sdec(cipher1, cek1) in”. Here, the section =m(A,B) signifies an equivalence check to confirm whether the decrypted message is equal to the message originally encrypted by A. This means, if the line instead were written as “let decryptedM = sdec(cipher1, cek1) in”, then regardless of whether or not decryptedM is equal to the original message, the script would continue executing and run the line bFinished(). Because of the equivalence check, bFinished() will only execute if the equivalence is true, and this gives assurance that the decryption process is valid. One can observe that processA() uses m(A,B) for the message when the cipher is created which is necessary for this equivalence check to function.

The event honest(party) is executed in the process honestUser and is used for multiple queries. Because of its execution in honestUser, it helps create a basis from which honest users—users that are not attempting to impersonate another user—can be distinguished for the purpose of querying. This event is used in what is arguably the most important query: “query A : party, B : party; event(honest(A)) && event(honest(B)) && attacker(m(A,B)).” This query checks the secrecy of the message m(A,B) by querying honest users and the “attacker” feature native to ProVerif. First, the two parties A and B are declared, and then the events honest(A) and honest(B) signify that both parties are honest users. Lastly, attacker(m(A,B)) determines whether the attacker can deduce the secret message between A and B or not. In summary, it checks if the attacker can retrieve a secret between two honest users, and the dishonestUser process separately checks if an honest user can be impersonated or tricked into unwanted communication.

The final two queries, “query A : party, B : party, keyAB : bitstring; event(honest(A)) && event(honest(B)) && event(viewEndB(A,B,keyAB)) ==> event(viewBeginA(A,B,keyAB))”, and “query A : party, B : party, keyAB : bitstring; event(honest(A)) && event(honest(B)) && inj-event(viewEndB(A,B,keyAB)) ==> inj-event(viewBeginA(A,B,keyAB))” check whether the two parties agree on the shared key cek or not. The viewBegin(A/B) events are executed by both parties once they create the key cek, and viewEnd(A/B) occurs at the end of each process. As discussed earlier in the paper, correspondence queries, signified by ==>, are used to determine the order of events. In essence, these events are checking if viewEndB occurs after viewEndA while also confirming that the values keyAB with parties A, B are in agreement between the two processes. The only difference between the queries is that the second query is injective, which checks if there is a one to one correspondence between the events.

With an explanation of the queries covered, it will now be helpful to examine the summarized output of the Proverif script:

```
Verification summary:

Query not event(aFinished) is false.

Query not event(bFinished) is false.

Query not (event(honest(A_1)) && event(honest(B_1)) &&
attacker_bitstring(m(A_1,B_1))) is true.

Query event(honest(A_1)) && event(honest(B_1)) &&
event(viewEndB(A_1,B_1,keyAB)) ==> event(viewBeginA(A_1,B_1,keyAB)) is true.

Query inj-event(viewEndB(A_1,B_1,keyAB)) && event(honest(A_1)) &&
event(honest(B_1)) ==> inj-event(viewBeginA(A_1,B_1,keyAB)) is true.
```

The first two events are false because they are checking if the event happens or not. Because queries are framed negatively with the word “not”, the double negative “not” and “false” means that aFinished and bFinished are true. For the next attacker query, it is stated that the query is “not…true” which means the attacker is not able to derive the secret message m(A,B). This query result is one of the major conclusions of this thesis because it is a proof that the secret message maintains secrecy within the protocol. The last two queries do not contain “not” because they are correspondence queries, and they are true which confirms that the key exchange of symmetric key cek occurs in an expected behavior.

Some small changes to the ECDH protocol, such as the transmission of fmk over a public channel, can lead to an attack. Appendix 5 provides a graphical trace of an attack if fmk is sent directly to B in the ECDH scheme. If partyA sends out (c, (cipher, fmk, gWithA)) instead of (c, (cipher, ckB, gWithA)), the attacker is able to use the constructor hkdf_expand(fmk) to reconstruct the symmetric key used for encryption as was discussed in section 4. The attack trace provides an explicit enumeration of this attack. In the attack, it is shown that the partyA sends (~M, ~M_1, ~M_2), and ~M_1 represents fmk. It is then stated that "the attacker has the message sdec(~M ,hfkdf_expand(~M_1)) = m(partyA_1,partyB_1). ~M represents the cipher text, so the input for the decryption is the cipher text and hkdf_expand(fmk) which is the same as cek. The decryption of these two values leads to m(A,B) which is the secret message. This serves as an example of how ProVerif can outline attacks, but this attack is not present in the ECDH scheme because fmk is not transmitted over a public channel.

Breaking down an attack into an attack trace gives an indication of how horn clauses play a role in ProVerif. In the case of encryption and decryption, for example, an attack can be thought of as whether an attacker has access to a set of variables. One paper [15] gives a generalization for how decryption can be presented through horn clauses. The clause "att(senc(m, k)) ∧ att(k) ⇒ att(m)" [15] gives the relationship between encryption (senc), a secret key k, and a message m. The "∧" sign is the logical sign for "and," so this clause states that if the attacker has the encrypted message resulting from m and k, and the attacker has k separately, then the attacker can recover m. The example just given about an attack with public fmk can be similarly reduced. att(senc(m,hkdf_expand(fmk)) ∧ att(fmk) ⇒ att(m) presents how fmk and encryption/decryption are related. Through this expression, one can see that the encryption process relies on the expansion of fmk, and because that is the symmetric key, it can also be used to decrypt.

It is also meaningful to look at clauses that do not hold true in the protocol. For example, in the correct protocol, att(senc(m),hkdf_expand(xor(ckB, dhKey))) ∧ att(ckB) ∧ att(dhKey) ⇒ att(m). This states that if the attacker has ckB and dhKey, then the attacker can derive the message m. However, because the attacker only has ckB and not dhKey, it is the case that this statement is false. The properties of the dhKey could be further expressed in a similar manner to show that the attacker does not have the necessary components to construct it. By reducing the actions of the protocol to these logical equivalences, it can be seen how ProVerif operates using horn clauses. Through examining which variables the attacker can access and the relationships between those variables, ProVerif can conclude whether or not an attack is present by testing all possible combinations.

5.2 Conclusions

The analysis of the protocols above has shown that confidentiality of the secrets m(A,B) is maintained. An attacker is not able to derive the message between two parties, and there is correspondence between the established symmetric key. It is therefore concluded that the protocols use of DH and RSA along with sending a XOR encrypted component for retrieval of the symmetric message encryption key is sufficient for maintaining secrecy of that message insofar as it has been represented within ProVerif.

6 Summary

This thesis has examined and analyzed an ECDH and RSA protocol created by the CDOC 2.0 project. The analysis was achieved by converting these protocols to pi calculus files that could be interpreted by ProVerif. In the conversion of these protocols, the cryptographic processes were abstracted to create a logical representation of the relationships between different constructors such as encryption and decryption. Some processes, like ECDH and RSA, were reinterpreted in a more general format due to this abstraction. The results have shown that, within the capabilities of ProVerif, an attacker is unable to derive a secret message exchanged between parties A and B for either cryptographic scheme. It has further been shown that there is one to one correspondence between the two parties' view of the symmetric key from when it is created to when the process completes. These results are validated by ProVerif itself due to its logically rigorous structure. ProVerif performs a complete reduction of all cryptographic primitives in the schemes to logically deduce if there are any vulnerabilities. It is assumed that the corresponding primitives themselves are secure, and under this assumption, attacks against the secrecy of the protocols have not been found.

References

[1] M. Oruaas and J. Willemson, “Developing requirements for the new encryption mechanisms in the Estonian Eid Infrastructure,” *Communications in Computer and Information Science*, pp. 13–20, 2020. doi:10.1007/978-3-030-57672-1_2

[2] B. Blanchet, "Automatic verification of security protocols in The symbolic model: The verifier proverif," Foundations of Security Analysis and Design VII, pp. 54–87, 2014. doi:10.1007/978-3-319-10082-1_3

[3] B. Meng, W. Wang, and W. Chen, “Verification of Resistance of Denial of Service Attacks in Extended Applied Pi Calculus with ProVerif,” *Journal of Computers*, vol. 7, no. 4, Apr. 2012, doi: https://doi.org/10.4304/jcp.7.4.890-899.

[4] J. Zhang, L. Yang, W. Cao, and Q. Wang, “Formal Analysis of 5G EAP-TLS Authentication Protocol Using Proverif,” *IEEE Access*, vol. 8, pp. 23674–23688, 2020, doi: https://doi.org/10.1109/access.2020.2969474.

[5] R. Bresciani and A. Butterfield, “ProVerif Analysis of the ZRTP Protocol,” *International Journal for Infonomics*, vol. 3, no. 3, pp. 306–313, Sep. 2010, doi: https://doi.org/10.20533/iji.1742.4712.2010.0033.

[6] R. Küsters and T. Truderung, "Reducing protocol analysis with XOR to the XOR-free case in the horn theory based approach," Proceedings of the 15th ACM conference on Computer and communications security, 2008. doi:10.1145/1455770.1455788

[7] R. Küsters and T. Truderung, "On the automatic analysis of recursive security protocols with XOR," STACS 2007, pp. 646–657. doi:10.1007/978-3-540-70918-3_55

[8] Ralf Küsters and Tomasz Truderung, “Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation,” *IEEE Computer Security Foundations Symposium*, Jul. 2009, doi: https://doi.org/10.1109/csf.2009.17.

[9] B. Blanchet and B. Smyth, "ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial," 2011.

[10] B. Blanchet, “Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif,” *Foundations and Trends® in Privacy and Security*, vol. 1, no. 1–2, pp. 1–135, 2016, doi: https://doi.org/10.1561/3300000004.

[11]“CDOC 2.0 spetsifikatsioon Tehniline dokument.” Accessed: Jan. 10, 2024. [Online]. Available: https://installer.id.ee/media/cdoc/cdoc_2_0_spetsifikatsioon_d-19-12_v1.9.pdf

[12] C. Shi and K. Yoneyama, “Verification of LINE Encryption Version 1.0 Using ProVerif,” I. Atsuo and Y. Kan, Eds., Springer International Publishing, 2018.

[13] Jean-Philippe Aumasson, *Serious cryptography : a practical introduction to modern encryption*. San Francisco: No Starch Press, 2018.

[14] M. Nemec, M. Sys, P. Svenda, D. Klinec and V. Matyas, "The Return of Coppersmith’s Attack: Practical Factorization of Widely Used RSA Moduli", *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[15] Bruno Blanchet. The Security Protocol Verifier ProVerif and its Horn Clause Resolution Algorithm. Electronic Proceedings in Theoretical Computer Science, 2022, 373, pp.14 - 22. ff10.4204/eptcs.373.2ff. ffhal-03897677f

[16] “Tamarin-Prover Manual Security Protocol Analysis in the Symbolic Model The Tamarin Team,” 2023. Available: https://tamarin-prover.com/manual/master/tex/tamarin-manual.pdf

[17] “CryptoVerif,” *bblanche.gitlabpages.inria.fr*. https://bblanche.gitlabpages.inria.fr/CryptoVerif/

[18]“EasyCrypt Reference Manual,” 2018. Accessed: Mar. 8, 2024. [Online]. Available: https://cs-people.bu.edu/gaboardi/teaching/S24-CS599/easycrypt-refman.pdf

[19] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 2020, doi: https://doi.org/10.1109/tit.1976.1055638.

[20] “ECDH Key Exchange - Practical Cryptography for Developers,” *Nakov.com*, 2021. https://cryptobook.nakov.com/asymmetric-key-ciphers/ecdh-key-exchange

[21] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 26, no. 1, pp. 96–99, Jan. 1983, doi: https://doi.org/10.1145/357980.358017.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Liam Simonos Warren

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “ProVerif Analysis of CDOC 2”, supervised by Nikita Snetkov
    1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
    1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

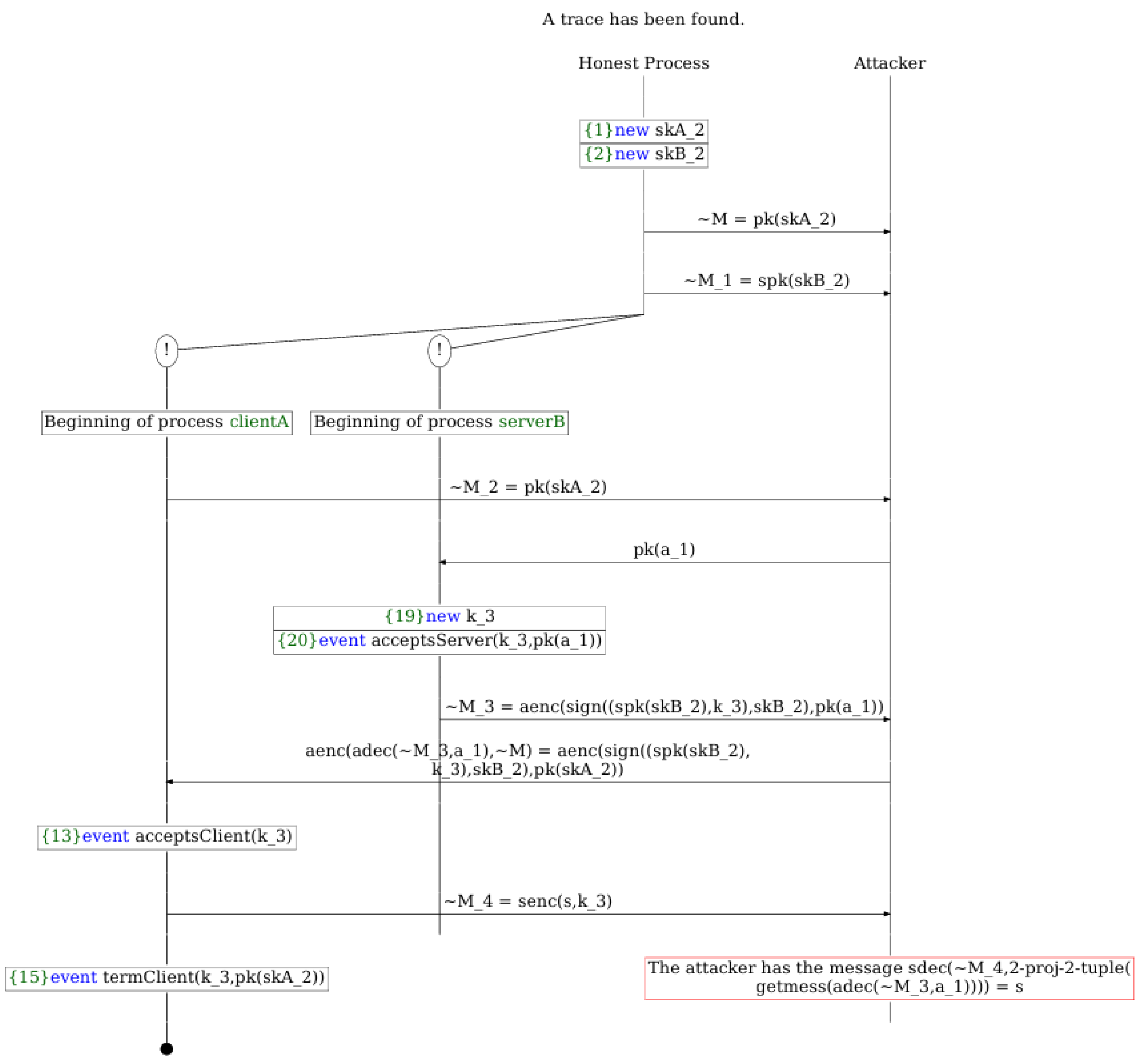
25/01/2024

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Handshake protocol in ProVerif

```
“(* Symmetric key encryption *)

type key.
fun senc(bitstring, key): bitstring.
reduc forall m: bitstring, k: key; sdec(senc(m,k),k) = m.

(* Asymmetric key encryption *)

type skey.
type pkey.

fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.

reduc forall m: bitstring, sk: skey; adec(aenc(m,pk(sk)),sk) = m.

(* Digital signatures *)

type sskey.
type spkey.

fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.

reduc forall m: bitstring, ssk: sskey; getmess(sign(m,ssk)) = m.
reduc forall m: bitstring, ssk: sskey; checksign(sign(m,ssk),spk(ssk)) = m.

free c:channel.

free s:bitstring [private].
query attacker(s).

event acceptsClient(key).
event acceptsServer(key,pkey).
event termClient(key,pkey).
event termServer(key).

query x:key,y:pkey; event(termClient(x,y))==>event(acceptsServer(x,y)).
query x:key; inj-event(termServer(x))==>inj-event(acceptsClient(x)).

let clientA(pkA:pkey,skA:skey,pkB:spkey) =
        out(c,pkA);
        in(c,x:bitstring);
        let y = adec(x,skA) in
        let (=pkB,k:key) = checksign(y,pkB) in
        event acceptsClient(k);
        out(c,senc(s,k));
        event termClient(k,pkA).

let serverB(pkB:spkey,skB:sskey,pkA:pkey) =
        in(c,pkX:pkey);
        new k:key;
        event acceptsServer(k,pkX);
        out(c,aenc(sign((pkB,k),skB),pkX));
        in(c,x:bitstring);
        let z = sdec(x,k) in
        if pkX = pkA then event termServer(k).

process
        new skA:skey;
        new skB:sskey;
        let pkA = pk(skA) in out(c,pkA);
        let pkB = spk(skB) in out(c,pkB);
        ( (!clientA(pkA,skA,pkB)) | (!serverB(pkB,skB,pkA)) )" [9]
```

Appendix 3 – Handshake protocol attack trace graphic

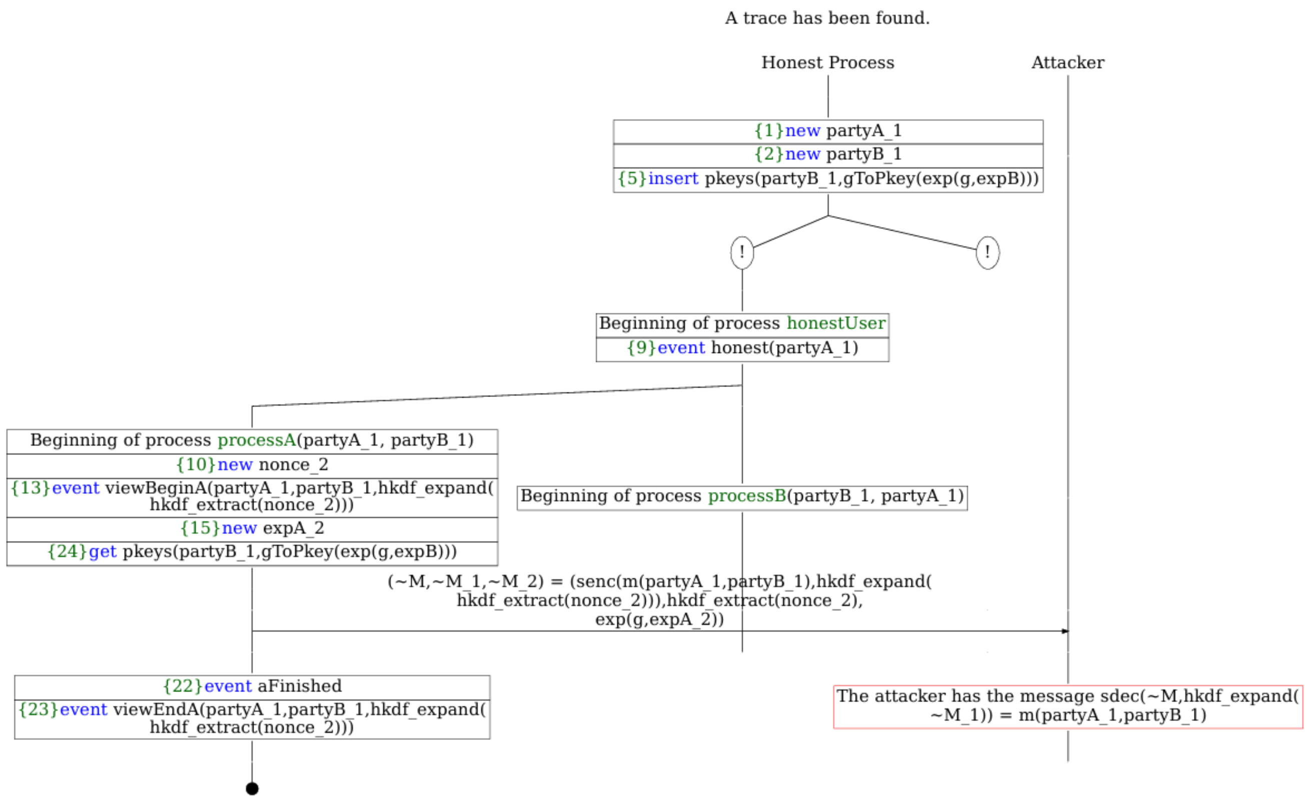

A trace has been found.
Honest Process
Attacker
{1}new skA_2
{2}new skB_2
~M = pk(skA_2)
~M_1 = spk(skB_2)
!
!
Beginning of process clientA
Beginning of process serverB
~M_2 = pk(skA_2)
pk(a_1)
{19}new k_3
{20}event acceptsServer(k_3,pk(a_1))
~M_3 = aenc(sign((spk(skB_2),k_3),skB_2),pk(a_1))
aenc(adec(~M_3,a_1),~M) = aenc(sign((spk(skB_2), k_3),skB_2),pk(skA_2))
{13}event acceptsClient(k_3)
~M_4 = senc(s,k_3)
{15}event termClient(k_3,pk(skA_2))
The attacker has the message sdec(~M_4,2-proj-2-tuple(getmess(adec(~M_3,a_1)))) = s

Appendix 4 – CDOC ECDH scheme in ProVerif

```
(* Types *)
set ignoreTypes = false.

type pkey.
type party.
type G.
type exponent.

free c:channel.

(* HKDF functions *)
fun hkdf_extract(bitstring): bitstring.
fun hkdf_expand(bitstring): bitstring.

(* DH *)
const expB: exponent [private].
const g: G.
fun exp(G, exponent): G.
equation forall x: exponent, y: exponent; exp(exp(g, x), y) = exp(exp(g, y), x).

(* XOR function [12] *)
fun xor(bitstring,bitstring):bitstring.
equation forall x:bitstring, y:bitstring; xor(xor(x,y),y)=x.
equation forall x:bitstring; xor(x,xor(x,x))=x.
equation forall x:bitstring; xor(xor(x,x),x)=x.
equation forall x:bitstring, y:bitstring; xor(y,xor(x,x))=y.
equation forall x:bitstring, y:bitstring; xor(xor(x,y),xor(x,x))= xor(x,y).
equation forall x:bitstring, y:bitstring; xor(xor(x,y),xor(y,y))= xor(x,y).

(* Symmetric encryption and decryption *)
fun senc(bitstring, bitstring): bitstring.
reduc forall m:bitstring, k:bitstring; sdec(senc(m,k),k) = m.

(* Type conversion *)
fun gToBitstring(G) : bitstring [data, typeConverter].
fun gToPkey(G) : pkey [data, typeConverter].
reduc forall g:G; pkeyToG(gToPkey(g)) = g.

(* secret b/t two parties *)
fun m(party,party) : bitstring [private]. (* private because attacker cannot derive secret
just from party names *)
```

```
(* Key table *)
table pkeys(party,pkey).

(* Queries & Events *)
event aFinished().
event bFinished().

event honest(party).

event viewBeginA(party,party,bitstring).
event viewBeginB(party,party,bitstring).
event viewEndA(party,party,bitstring).
event viewEndB(party,party,bitstring).

query event(aFinished()).
query event(bFinished()).


(* attacker tries to guess secret message *)
query A : party, B : party; event(honest(A)) && event(honest(B)) &&
attacker(m(A,B)).

query A : party, B : party, keyAB : bitstring; event(honest(A)) && event(honest(B))
&& event(viewEndB(A,B,keyAB)) ==> event(viewBeginA(A,B,keyAB)).

query A : party, B : party, keyAB : bitstring; event(honest(A)) && event(honest(B))
&& inj-event(viewEndB(A,B,keyAB)) ==> inj-event(viewBeginA(A,B,keyAB)).

let processA (A: party, B : party) =

        (* Generating CEK from FMK using HKDF *)
        new nonce:bitstring;
        let fmk = hkdf_extract(nonce) in
        let cek = hkdf_expand(fmk) in
        event viewBeginA(A,B, cek);

        (* Message encryption *)
        let cipher = senc(m(A,B), cek) in

        (* DH encapsulation *)
        new expA : exponent;
        let gWithA = exp(g, expA) in

        (* Retrieves PK of B *)
        get pkeys(=B, pkB: pkey) in

        (* DH Completion *)
        let dhB = pkeyToG(pkB) in
        let gWithAandB = exp(dhB, expA) in (* encapsulation *)
        let dhKey = gToBitstring(gWithAandB) in

        (* A XORs *)
        let ckB = xor(fmk, dhKey) in

        (* Sends info to B *)
        out(c, (cipher, ckB, gWithA));

        event aFinished();
        event viewEndA(A,B, cek);
        0.

let processB (B : party, A : party) =

        (* Receive info from A *)
        in (c, (cipher1:bitstring, ckB1: bitstring, gWithA: G));

        (* Decapsulation *)
        let gWithBandA = exp(gWithA, expB) in

        (* DH completion *)
        let dhKey = gToBitstring(gWithBandA) in

        (* XOR and cek derivation *)
        let fmk1 = xor(ckB1, dhKey) in
        let cek1 = hkdf_expand(fmk1) in
        event viewBeginB(A,B, cek1);

        (* Decryption *)
        let (=m(A,B)) = sdec(cipher1, cek1) in (* checks if the decryption gives original
message *)
        event bFinished();
        event viewEndB(A,B, cek1);
        0.

let honestUser(A: party, B: party) =

        event honest(A);
        (processA(A, B))
        |
        (processB(B, A)).

let dishonestUser =
        new name: party;
        in(c, (expX : exponent));
        let dhX = exp(g, expX) in
        let pkX = gToPkey(dhX) in

        insert pkeys(name,pkX);
        out (c, (name,pkX));
        0.

(* Main process *)
process
        new partyA: party;
        new partyB: party;
        let dhB = exp(g, expB) in
        let pkB = gToPkey(dhB) in
        insert pkeys(partyB, pkB);

        (!honestUser(partyA, partyB) | !dishonestUser)
```

Appendix 5 – CDOC ECDH attack trace example



A trace has been found.
Honest Process
Attacker
{1}new partyA_1
{2}new partyB_1
{5}insert pkeys(partyB_1,gToPkey(exp(g,expB)))
!
!
Beginning of process honestUser
{9}event honest(partyA_1)
Beginning of process processA(partyA_1, partyB_1)
{10}new nonce_2
{13}event viewBeginA(partyA_1,partyB_1,hkdf_expand(hkdf_extract(nonce_2)))
{15}new expA_2
{24}get pkeys(partyB_1,gToPkey(exp(g,expB)))
Beginning of process processB(partyB_1, partyA_1)
(~M,~M_1,~M_2) = (senc(m(partyA_1,partyB_1),hkdf_expand(hkdf_extract(nonce_2))),hkdf_extract(nonce_2),exp(g,expA_2))
{22}event aFinished
{23}event viewEndA(partyA_1,partyB_1,hkdf_expand(hkdf_extract(nonce_2)))
The attacker has the message sdec(~M,hkdf_expand(~M_1)) = m(partyA_1,partyB_1)

Appendix 6 – CDOC RSA scheme in ProVerif

```
(* Types *)
set ignoreTypes = false.

type skey.
type pkey.
type party.

free c:channel.

(* HKDF functions *)
fun hkdf_extract(bitstring): bitstring.
fun hkdf_expand(bitstring): bitstring.

fun pk(skey): pkey.

(* XOR function [12] *)
fun xor(bitstring,bitstring):bitstring.
equation forall x:bitstring, y:bitstring; xor(xor(x,y),y)=x.
equation forall x:bitstring; xor(x,xor(x,x))=x.
equation forall x:bitstring; xor(xor(x,x),x)=x.
equation forall x:bitstring, y:bitstring; xor(y,xor(x,x))=y.
equation forall x:bitstring, y:bitstring; xor(xor(x,y),xor(x,x))= xor(x,y).
equation forall x:bitstring, y:bitstring; xor(xor(x,y),xor(y,y))= xor(x,y).

(* Public key encryption and decryption *)
fun aenc(pkey, bitstring): bitstring.
reduc forall m:bitstring, k:skey; adec(k, aenc(pk(k),m)) = m.

(* Symmetric encryption and decryption *)
fun senc(bitstring, bitstring): bitstring.
reduc forall m:bitstring, k:bitstring; sdec(senc(m,k),k) = m.

(* secret b/t two parties *)
fun m(party,party) : bitstring [private]. (* private because attacker cannot derive secret
just from party names *)

free skB : skey [private].
free kekB : bitstring [private]. (* Not generated because only 1 B party and no reverse
function *)

(* Key table *)
```

```
table pkeys(party,pkey).

(* Queries & Events *)
event aFinished().
event bFinished().

event honest(party).

event viewBeginA(party,party,bitstring).
event viewBeginB(party,party,bitstring).
event viewEndA(party,party,bitstring).
event viewEndB(party,party,bitstring).

query event(aFinished()).
query event(bFinished()).

(*attacker tries to guess secret message *)
query A : party, B : party; event(honest(A)) && event(honest(B)) &&
attacker(m(A,B)).

query A : party, B : party, keyAB : bitstring; event(honest(A)) && event(honest(B))
&& event(viewEndB(A,B,keyAB)) ==> event(viewBeginA(A,B,keyAB)).

query A : party, B : party, keyAB : bitstring; event(honest(A)) && event(honest(B))
&& inj-event(viewEndB(A,B,keyAB)) ==> inj-event(viewBeginA(A,B,keyAB)).

let processA (A: party, B : party) =

        (* Generating CEK from FMK using HKDF *)
        new nonce:bitstring;
        let fmk = hkdf_extract(nonce) in
        let cek = hkdf_expand(fmk) in
        event viewBeginA(A,B, cek);

        (* Message encryption *)
        let cipher = senc(m(A,B), cek) in

        (* A XORs *)
        let ckB = xor(fmk, kekB) in

        (* Retrieves PK of B *)
        get pkeys(=B, pkB: pkey) in

        (* PK encapsulation *)
        let capsB = aenc(pkB, kekB) in

        (* Sends info to B *)
        out(c, (cipher, ckB, capsB));

        event aFinished();
        event viewEndA(A,B, cek);
        0.

let processB (B : party, A : party) =

        (* Receive info from A *)
        in (c, (cipher1:bitstring, ckB1: bitstring, capsB1: bitstring));

        (* Decapsulation *)
        let kekB1 = adec(skB, capsB1) in

        (* XOR and cek derivation *)
        let fmk1 = xor(ckB1, kekB1) in
        let cek1 = hkdf_expand(fmk1) in
        event viewBeginB(A,B, cek1);

        (* Decryption *)
        let (=m(A,B)) = sdec(cipher1, cek1) in (* checks if the decryption gives original
message *)  event bFinished();
        event viewEndB(A,B, cek1);
        0.

let honestUser(A: party, B: party) =

        event honest(A);
        (processA(A, B))
        |
        (processB(B, A)).

let dishonestUser =
        new name: party;
        in(c, skX : skey);
        let pkX = pk(skX) in

        insert pkeys(name,pkX);
        out (c, (name,pkX));

        0.

(* Main process *)
process
        new partyA: party;
        new partyB: party;
        let pkB = pk(skB) in (* Creates key pair for B *)
        insert pkeys(partyB, pkB);

        (!honestUser(partyA, partyB) | !dishonestUser)
```