Cryptographic protocols systematic analysis: the importance of tuple bracketing

Marceau DIDA
Student - Software Engineering
TÉLÉCOM Nancy, Villers-lès-Nancy, FRANCE
Email: dida.marceau@gmail.com

Ambroise BAUDOT
Student - Internet Systems and Security
TÉLÉCOM Nancy, Villers-lès-Nancy, FRANCE
Email : ambroise.baudot@gmail.com

***Abstract*—The world of cryptographic protocols is a very complex one and it is drove with the motivation that the protocols should be safe to use. In order to prove this, the tools of formal verification have been largely developed throughout the last few decades, some having really interesting results. The formal specification of protocols needs to follow certain rules according to the tools used. But what exactly is the impact of a small change in this specification on the verification ? How does the mere modification of bracketing of a tuple can have any impact on a protocol safety ? Is it possible to find new attacks on known protocols with this little change in the verification process ? The major conclusion of this paper is that modifying the bracketing of tuple can have a strong impact on the outputs given by a tool such as ProVerif. It can actually allow us to discover new attacks on famous protocols or even more frequently cause the tool used to take up to an infinite time to finish the verification.**

***Index Terms*—cryptography, protocol, safety, ProVerif, analysis, bracketing**

In this paper will be treated the subject of bracketing modification inside the specification of cryptographic protocols. First, the context will be introduced in a first part, then the importance of the tuple bracketing will be shown. A third part will present the formal verification tool used for this work : ProVerif. Then, our work will be described and explained and finally a last part will help to conclude this paper.

I. INTRODUCTION

A. Global context

The internet has drowned our world. And with it the networks have been developing at a very fast rate. They are everywhere. More and more information are exchanged throughout this global network, from the most trivial material between mere friends to the most crucial one between states. It is essential to keep those information away from the knowledge of what could be malicious persons or organisation. In others words, the networks need to be secured. The security of networks is a complete subject of study on its own as there as many ways to secure a network as there are ways to go around this security.

This problem goes beyond the simple technical and physical organisation of the networks. Indeed, the ways that one sends or receives the messages on the network is also of an extreme importance. Throughout the development of these exchanges it has been realised that in order to have a better understanding of how the messages were exchanged and how some were caught by malicious users. Norms and and standards were needed. These set of rules are what is called protocols. To be a bit more specific, there are different kind of protocols and the one that are interesting for the safe exchanges of messages between machines, are cryptographic protocol. Indeed, one will try and encrypt the information before sending it to the user and on the other end of the communication the user will decrypt the information thus accessing it and assuring its confidentiality. Historically these protocols started very simple and are now complex algorithms that are being used to secure and insure that all the data stay hidden. As the threat grew bigger and bigger, it is in the end of the 1970's that protocols started to be created (some of the first ones being : Needham-Schroeder [7], Kerberos [8], Otway-Rees [11], Yahalom [5]) and we are now using far more complex protocols (SSL [6], TLS [13] which are protocols that still being updated up to now), this complex protocols are used to secure most of the exchanges on the Internet, and can for instance be used for the e-voting.

To give a bit of insight, we shall study one of the oldest protocol there is. It was invented by Roger Needham, and Michael Schroeder, british and american computer scientists, and described in a paper published in December 1978 [7].

The simplest form of the Needham-Schroeder protocol can be expressed by those three exchanges only (Fig. 1).

Fig. 1. Description of the Yahalom Protocol

$$\begin{array}{lcl} A \to B & : & \{A, N_A\}_{pk(B)} \\ B \to A & : & \{N_A, N_B\}_{pk(A)} \\ A \to B & : & \{N_B\}_{pk(B)} \end{array}$$

With A and B two machines willing to exchange a message: N_B. For all the followings examples we are going

to consider that this N_B value and all the others N values are simple numbers but in fact it could be any kind of data that needs to be transferred from A to B. We call them nonce: a randomly generated number created by a machine for a single usage. They are used as the session IDs. Moreover we can note that $pk(X)$ means that we are access the public key of the machine X. And last, the notation $\{n\}_{pk(X)}$ is used to express that n is encrypted with the public key of X.

In this simple example of the Needham-Schroeder protocol we only illustrate the actual communication between the two machines but in reality, there are other operations that need to be done in order to properly set up the communication. Indeed, each machine needs to know the public key of the other (pk). The process of this key exchange can be tricky and has a lot of possible ways of treating it. To simplify the understanding of the subject, it will be accepted with all the protocols that the machines A and B know that keys from one another.

Finally, one machine must reveal on the public channel its will to communicate with another. That is done by simply exchanging on the channel a message with A, B. This allows the two machines to be prepared for the communication to come.

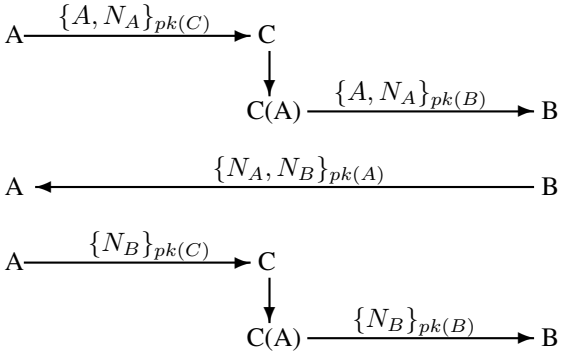
On the protocol itself, you can see that it is fairly simple. Indeed, A (namely Alice) wants to communicate with B (Bob) to identify herself. This authentication process will be possible thanks to a nonce N_B created by B. At first, she sends a message with her identity and another nonce N_A. This message is obviously encrypted with the public key of B that A got earlier on. B receives this message and generates the nonce N_B. He then sends his response with N_A as well as N_B. He hides this exchange by encrypting it all with the public key of A. At this point, A has received the data that she wanted, that is to say N_B. But only to confirm that the exchange went as expected she sends a last message to B with the N_B nonce only. B is now sure that A received the information safely. One could think that with this set of rules it would be hard to intercept the N_B value. As indeed, all the exchange is encrypted properly with the public key of the recipient and thus completely hidden from any machine apart from A and B.

However, there is a possibility for a malicious machine to actually take hold of the message by a well known type of attack called man-in-the-middle.

The point of the intruder is to impersonate another machine. It will be able to impersonate the machine A from machine B's perspective thus making the machine B believe that it is actually A, fooling completely the authentication process.

In the next illustration we will show you how an intruder named C can get access to the value N_B without any of the other two machines involved realising it. And thus showing how the simple protocol that is Needham-Schroeder can be attacked in order to get access to a specific value or to impersonate another machine.

Fig. 2. Description of the attack of the Needham-Schroeder Protocol



***Attack :** B is speaking to C instead of A.*

Here it can be seen that as before, A tries to communicate with B, following the exact Needham-Schroeder protocol as defined above. But in this case, an intruder C (Charlie), takes the message and sends it to B. As it should, Bob responds by sending the correct message encrypted with the public key of A, but he sends it to C as he is the one that started the exchange and B has no way to know that information. Obviously, C cannot decipher the message that he has just received so he transfers it, unchanged, to $Alice$. Finally, following the same pattern, A sends the final message to C, believing that it is B, and C simply transfers it and the attack is completed.

At the end of this process, C managed to fool B into believe that he is A and A into believing that he is B. The malicious part of the attack is that both A and B are not aware that the authentication process failed, but they are actually believing that everything went how it was suppose to which is even more dangerous. The effects of such an authentication misbehaviour can be dramatic as in further exchanges both A and B will be mistaken and will send their information to the wrong machine.
This attack was discovered by Gavin-Lowe (see [9]).

This allows us to illustrate how protocols, even though they aim at securing the exchanges on a network, have downsides. Sometimes it is completely independent from the protocol and only based on the network's limitations. Although, this protocol stays very simple it shows something that is common to all the security protocols created through history, even the most recent ones. And tools will be needed to help find those attacks and eventually prove that a protocol is safe or not.

B. Existing tools

We will see now that there are a lot of tools available. Indeed, creating a software that is able to help on this matter is a subject that interested a lot in the past decades. This software is a formal verification software. To demonstrate that a protocol is safe, or to proof that it is not safe (*i.e.* finding

an attack), the protocol has to approximate the protocol. It is worth noticing that the issue of a protocol's safety is an undecidable problem [10], it is not certain that the software can find an sure answer (like safe or not safe). It sometimes means that with some protocols, the verification tools won't be able to give an answer as to whether or not a protocol is safe. The possible outputs depend on the tool used, and in our case they are detailed in section III.B. So to put it in a nutshell, the tool aims at finding if a specified protocol is safe, and if not, it should show the entire process that led to an attack.

As stated above, this topic is really wide and it is unthinkable to create a software that would be able to solve all the problems. Therefore, there are multiple software that currently exist. The most famous software are ProVerif [4], Tamarin Prover [1]. Otherwise, software for more specific applications include : AVISPA [14], CryptoVerif [2], ...

Throughout our work we used the ProVerif software.

C. Subject of study

Now that the general context is set, our aim, when working on this subject was to understand the effects of tuple bracketing inside the specifications of the security protocols. Indeed, as seen with the Needham-Schroeder protocol example, many protocols need to exchange multiple values to their counterpart. This data, when designed inside of a computer needs to be represented inside a data structure. For the most part, software use tuples for this representation. Basically, a tuple is a list of values one after the other. Obviously a tuple structure needs bracketing. And it has been noticed that this bracketing is really important in the specification of a protocol, as indeed, the original specification is modified when a different bracketing is used. This difference of specification is not proven to be safe, and as we will see throughout this whole report, it can have dramatic changes on a protocol and its potential attacks. Our subject of study was the creation of a program that would allow us to verify whether or not modification of bracketing inside the specification of protocols would have any impact on its safety, and if so, understand why and report it.

D. Contribution

During the period of our work, we had to first understand these notions of security protocols, specification, attacks, and ProVerif, the tool used to verify the security of protocols. We were at the same time introduced to the important matter of tuple bracketing. We then, following the terms of the subject of study, designed a lexical parser for the ProVerif language whose aim was to mark all the tuples inside of a given ProVerif program used to specify a security protocol. Doing so, we created and generated the complete list of possible bracketing of this particular file, and then looked through all of those, searching for any difference between the original output and the one of the generated files.

As a result, we discovered an attack to the Yahalom security protocol. It is a success in itself as it was not sure at all that the protocols we studied were impacted in any way by this change of bracketing, but we managed to prove that one was. And even more significantly, this attack on the Yahalom protocol was not listed in any of the specific literature about the subject. This finding leaves hope to find more and more attacks on other protocols and in doing so, finding a way to prevent them and look towards safer exchanges of information.

II. The importance of bracketing

A. Some more case study with the Needham-Schroeder protocol

Going back to the Needham-Schroeder protocol, an attack was discovered on the standard version of the protocol. As stated above, this attack causes the security of the protocol to be strongly affected, and the protocol needs to be changed if it aims at being used in the future.

Gavin Lowe after finding the above attack, invented a modified version of Needham-Schroeder protocol (see [9]). This protocol was designed to prevent the man-in-the-middle attack. As indeed, we explained that this attack was possible because the intruder impersonates one of the machines. Thus, Lowe explains that by adding the identity of the machine inside the protocol itself, this kind of attack is not possible anymore.

That gives us the modified version of the Needham-Schroeder protocol known as Needham-Schroeder-Lowe :

Fig. 3. Description of the Needham-Schroeder-Lowe protocol

$$\begin{array}{lcc} A \to B & : & \{A, N_A\}_{pk(B)} \\ B \to A & : & \{B, N_A, N_B\}_{pk(A)} \\ A \to B & : & \{N_B\}_{pk(B)} \end{array}$$

Here, with the B value added to the second message of the exchange, A is now sure that B is actually being the one talking to it.

What is going to interest us next is this second message. Here we see that there are three values that are being exchanged and we need to represent it with a tuple structure. Gavin Lowe, when he decided to add this identity value, chose to put this value on the left of the tuple. That leaves us with the tuple : $\{B, N_A, N_B\}$. But what would happen if this value was put in another position inside the tuple? One further question is how to represent this tuple. We saw that a machine needs to represent the tuple and according to the tool used, the tuple can be represented by a three-element tuple, or by a combination of doubles.

With those two questions at hand, we are left with the possibilities illustrated on the Figure 4.

It is important to notice that here is listed the complete set of combinations and bracketing coming from the addition of the identity value. So for each possible position in the tuple where it can be added, we derivate it into all the possible bracketings. Indeed, it is possible to add the identity value to the left of the tuple, to the right and finally to the middle, leaving us with three combinations that we need to derivate with their corresponding bracketing possibilities. As it will be explained later, for a tuple of three elements we can prove

Fig. 4. Possible combinations after adding an identity value to the second message of Needham-Schroeder-Lowe

$$\begin{array}{ll} (N_A, (N_B, B)) & (1) \\ ((N_A, N_B), B) & (2) \\ (N_A, (B, N_B)) & (3) \\ ((N_A, B), N_B) & (4) \\ (B, (N_A, N_B)) & (5) \\ ((B, N_A), N_B) & (6) \end{array}$$

that the number of possible bracketings is actually only two. Giving us the six possible combinations above.
Throughout our work, we did not focus on the addition of a new value. The main purpose of our work was to generate the possible bracketings from an already complete tuple, thus, never adding a new value to it before generating. In this particular example, the combinations (1) and (2) would be generated by our program assuming that the identity value is placed on the right of the tuple (Figure 5).

Fig. 5. Modified version of the Needham-Schroeder-Lowe protocol leading to an attack

$$\begin{array}{lcc} A \to B & : & \{A, N_A\}_{pk(B)} \\ B \to A & : & \{N_A, (N_B, B)\}_{pk(A)} \\ A \to B & : & \{N_B\}_{pk(B)} \end{array}$$

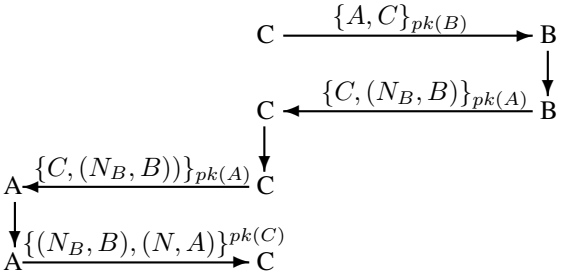
B. Consequences

Now, that we explained that a tuple can actually be bracketed in different ways, it is important to know what are the consequences of it on the security of protocols. To do so, we can use a formal verification tool such as ProVerif, to check the results on the possible bracketings.

It has been found that an attack exists on some of the possibilities (namely (1) and (3)). This attack is described on Figure 6.

As you can see from the description, C can still impersonate A but this time it is through the N_A value. Indeed, as we said before this value is not necessarily a number and we have the proof right here as N_A equals C. With this small little change the machine A now thinks that it has been talking with C ever since the beginning. Based on this belief, it sends him the useful information encrypted with the intruder's public key.

Fig. 6. Description of the attack of the Needham-Schroeder Protocol using a bad bracketing



***Attack :** A is speaking to C instead of B.*

Here it is another case of a man-in-the-middle attack, but this time it is not based on a weakness of the protocol itself but only based on an implementation detail that is the bracketing choice.

Remark :

In order to bring a correction to this attack there is no other choice than to force the bracketing of the tuple. But this time, using a bracketing that will not link the identity value with the important information. In our examples, that is to say B and N_B. If we take a look at all the possible bracketing for the protocol, we can in fact choose from all the possibilities except for the number (1) and (3).

We, then, based on the example of the Needham-Schroeder protocol, understand the actual importance of the bracketing. Indeed, behind tuples can hide thing such as security attacks that can eventually lead to very high risk for those who use this protocol. That is why, the work that we undertook is of a great importance as it might allow us to find attacks on protocol that are yet to be found.

III. ProVerif tool presentation

In this chapter we are going to take a closer look at the tool that we used to work on this bracketing issue. It will allow us to understand in a better way what is a tuple in ProVerif and how it is defined. We will also look at the security properties that are verified by the tool and finally the outputs given by ProVerif to understand if a security breach exists.

A. Language syntax

It is now time to dive into the ProVerif tool. This tool uses a language to help describe security protocols. To do so, it needs proper expressions, with defined keywords and syntax. A formal tool used to describe such a language is called a grammar and as with all the computer language, ProVerif uses a grammar. As the language is quite complex, the associated grammar is as difficult. To help with the understanding of the language a simplified version of the grammar is shown below (Figure 7).

Fig. 7. Simplified version of the ProVerif grammar from [3]

$M, N ::=$	$terms$
x, y, z	$variable$
a, b, c, k, s	$name$
$f(M_1, ..., M_n)$	$constructor\ application$
$D ::=$	$expressions$
M	$term$
$h(D_1, ..., D_n)$	$function\ application$
$fail$	$failure$
P, Q	$processes$
$\mathbf{0}$	nil
$out(N, M); P$	$output$
$in(N, x : T); P$	$input$
$P \mid Q$	$parallel\ composition$
$!P$	$replication$
$new\ a : T; P$	$restriction$
$let\ x : T = D\ in\ P\ else\ Q$	$expression\ evaluation$
$if\ M\ then\ P\ else\ Q$	$conditional$

There are a few points that are worth noticing in this grammar. Here is a list of those points with the explanations of what is their purpose inside the ProVerif language.

- **process :** Structure and keyword used to specify in general what are going to be the action executed for this protocol. Inside of it are usually created all the global data that needs to be created beforehand, for instance, the authentication keys of the machines. It also always to specify which functions are going to be called in thus initialising the protocol execution.

process

```
new Kas: key; new Kbs: key;
insert keys(A, Kas);
insert keys(B, Kbs);
(
 (!processInitiator(A, Kas)) |
 (!processResponder(B, Kbs)) |
 (!processS) |
 (!processK)
)
```

Here we see an example in an actual ProVerif file, of what a process statement can look like. With, as it can be seen, the keys creation and initialisation of the functions stating the execution of a protocol.

- **let :** This keyword is important as it has many uses. First, it allows to create an environment like a function. Often times, it is used to create a function to state the exact and detailed execution of a process inside a security protocol. For instance, in Needham-Schroeder protocol, ProVerif will need (in a simple version of the protocol) a function with a let statement to describe the behaviour of the machine A and another to describe machine B's.

```
let clientA(A:host, B:host,pkA:pkey,
    skA:skey,pkB:pkey,n:bitstring) =

 out(c,aenc((A,n),pkB));
 in(c,x:bitstring);
 let (=n,nn: bitstring) = adec(x,skA) in
 out(c, aenc(nn,pkB)).
```

- **event :** Keyword used to specify a specific point in the execution of the protocol specification. Indeed, when modeling the protocol inside ProVerif we sometimes want to be able to trigger some events after they have happened. This event keyword aims at expressing some events corresponding to the security properties checked during the protocol verification.

```
event beginBparam(host, host).
event endBparam(host, host).
event beginAparam(host, host).
event endAparam(host, host).


...

let processInitiator(pkS: spkey, ...) =
 in(c, (xA: host, hostX: host));
 if xA = A || xA = B then
 let skxA =
 if xA = A then skA else skB in
 let pkxA = pk(skxA) in
 (* Real start of the role *)
 event beginBparam(xA, hostX);
 out(c, (xA, hostX));
 ...
 event beginBfull(xA, hostX, pkX, ...);
 ...
 if hostX = B || hostX = A then
 event endAparam(xA, hostX);
 event endAfull(xA, hostX, pkX, ...);
 out(c, sencrypt(secretANa, Na));
 out(c, sencrypt(secretANb, NX2)).

...
```

In the extract from a ProVerif file are first the declaration of the events that are going to be used further in the specification. Then, inside the function (keyword `let`) describing the behaviour of the process initiator, some of these events are called again in specific position of the specification. The position corresponds to specific point in the execution of the protocol where a special event has been done.
For instance, $event\,beginBparam$ means that at this point of the protocol, the event corresponding to this description has been done. The event keyword is purely informative and has no effect on the specification of the protocol itself.

- **new :** Keyword used to create a structure. This structure can be a variable of any type. In most of the examples, we can notice that nearly all the keywords `new` are used to define and create new nonces (see the definition of a nonce in the introduction).

```
new Nb: nonce;
```

- **query :** Keyword used to specify the security properties that ProVerif needs to verify during its execution. If the query is verified then no intruder can breach this property, if not then ProVerif found some way for an intruder to break this particular security property using the specification of the protocol.

```
query attacker(secretANa);
      attacker(secretANb);
      attacker(secretBNa);
      attacker(secretBNb).
```

- **out, in :** Keywords used to specify that a particular process needs, at this point in the protocol, to send (out) or receive (in) a message on the global channel, representing the network. It, most of the times, needs to be followed by a specific format of information, amongst those are single variable, tuples, functions calls, etc ...

In the example used for the `let` expression we can clearly see an example of simple exchanges. This piece of ProVerif code is actually the specification for the machine A in the simple Needham-Schroeder protocol that we

used as an example in the introduction. We can see that it matches exactly the output and the inputs stated by the protocol specification itself.

As stated in the chapters above, an expression that is extremely important in our work is the tuple structure. With the study of the simplified grammar, we saw that ProVerif offers a lot to describe a protocol, amongst all the keywords and structure, ProVerif also allows to write special expressions like tuples and operations.

Therefore, there is a rule in the grammar to express the syntax of tuples. This needs a brief explanation of the notion of terms in ProVerif. A TERM is an expression used in all the ProVerif logical operations. And as any operation, this rule must be recursive and call itself on the right and left parts of the operator in question. Here, in ProVerif, there are three different type of terms. The first one is called TERM, the second PTERM and the last one GTERM. Each one of them allows the grammar to derivate towards specific operations and of course, have the same property of recursivity and priority of the operators.

Now, a tuple is simply a sequence of terms. So for each type of term stated above, there is a matching rule allowing to generate tuples. For example, the rule generating tuples for a TERM term is specified in Figure 8 extracted from the Proverif manual.

Fig. 8. Grammar rule for a tuple of terms extracted from the ProVerif manual [4]

$$\langle term \rangle ::= (seq\langle term \rangle)$$
$$with\, seq\langle X \rangle = \langle X \rangle, ..., \langle X \rangle$$

Now that we know exactly what a ProVerif tuple is, we can take a closer look at the security properties that the language can verifying and especially the ones that are going to interest us during our working process.

B. Security properties

To summarize, a file is written in the ProVerif language following a formal grammar explained above (complete examples of ProVerif protocols specification in Annex A). The software then tries to execute a set of derivations leading to an attack. The attacker has to breach a security feature. In this part, we will explain what are those features and those properties of security.

In practice, the user must specify which data ProVerif must check (for instance "The channel must not know that information", considering the channel as the attacker). To be clear, that is to say, for example, that the channel cannot know the id of the sessions, which would break the authentication process, and the protagonists must only send messages to the correct machines. The notion of correct machines is here used to describe a machine that is not being impersonated by any other machine.

Properties checked: We would like to know if a protocol is safe, but we cannot simply ask "*Is this protocol safe?*" to the ProVerif software. In fact, it depends on which security property is defined in the specification. Let's present briefly three main properties often verified : secret, authentication, and non-interference. These properties are declared inside a ProVerif program using the query keyword, explained earlier on.

- **Secret** : In a conversation, the secret must be preserved. For instance, when Bob and Alice are talking, it must not be possible that Eve can understand the messages. As we have said, we can consider that the attacker (Eve) is the channel, *i.e.* an attacker know all these public information which are the information exchanged that anyone can read. So for instance it is possible to implement in the protocol an encrypted exchange from Alice to Bob, called `secret`, and a specification of the secret query could be "The channel must not be able to know `secret`".
- **Authentication** : The authentication security query states that all users must be right about the identity of the machine they are addressing when sending and receiving a message. In other words, the intruder must not be able to impersonate any other machine.
- **Non-interference** : The non-interference is a property that guaranties that it is not possible to know where the messages are from. The content of the message could be public, but other properties are not distinguishable among several messages. Let's study an example where the property of non-interference is important. That is the case of anonymity. For instance with e-voting, it would have no purpose to verify the secret of the data, as it is known by everyone. (yes, no or name of a candidate in a vote for example), but it is important to ensure the anonymity of the person that voted. That is of course far from all there is to know of the non-interference, but it is enough to help understand it and the use that is made of it in our work. Indeed, some of the ProVerif files that we will come to test are actually verifying the non-interference property and it is important to understand it to appreciate the impact of it on the safety verification of the protocol.

C. Outputs

Once the ProVerif program is executed, it gives an output that the user has to thoroughly read in order to see whether the protocol specified in the program is actually safe or not. Listed underneath, are all the possible outputs given by ProVerif with the interpretation that can be made of them.

- The protocol is analysed as **"safe"** : "ProVerif" has not found any derivation that allows the attacker to break one of the specified query. The queries are marked as "true", since they are considered sound. The protocol is considered safe by ProVerif. However, that does not necessarily means that the protocol is actually safe. Indeed, ProVerif uses a particular formal model to verify the safety of the

protocols and some of the possible attacks are not tested at all during the verification. What is sure with this output is that ProVerif cannot find a way to break the properties specified in any of the queries.

- The protocol is analysed as "not safe" when **"a trace has been found"**. A trace is the sequence of events which leads to a security breach inside the protocol. That means that the software manages to build this sequence of events, and the protocol is necessarily not safe.
  In this kind of output, there is the detail of messages passing through the channel leading to the attack, and a link on each message to the corresponding operation of the protocol (there are not all the messages but with the protocol it is possible to find whose message it is, the recipient, why this message was sent and what will be the answer). It is possible to reconstruct the attack from the sequence of events, being the dialogue between the protagonist.
- The security of the protocol **"cannot be proved"**. It is importance to note that, as we have said, formal verification is undecidable. That is why to resolve these issues ProVerif make approximations, and when something looks like an attack, the tool tries to find the trace leading to the actual attack. But sometimes it cannot find this trace, and that is a possible reason why the safety of the protocol "cannot be proved". Or sometimes, there is nothing looking like an attack, but ProVerif is not able to demonstrate that the protocol is safe. That is the other possible reason for the "cannot be proved" output.

- **"TimeOut"** is not really an output of ProVerif, but it is possible that a protocol verification takes up to an infinite time to end as the problem is undecidable. For instance, big protocols like TLS can be running for weeks, but for "little protocols" (with very simple processes and a little number of rules used), if the material resources are continually increasing, it is possible thanks to heuristic rules to conjecture that the program will not stop. For our work purposes we took a time out of five minutes, considering that the original files took few milliseconds at most to be executed, we can say that if the execution did not end in five minutes, then there is a good chance that is will never end. Although, of course, we cannot affirm this for certain. We firstly have made our tests with `TimeOut=30` seconds, because running on all our files spent between four and six hours (on a personal computer, with 8Go RAM, core i7-7th, SSD). Indeed, most of the files do not take more than one minute to be executed, and for this kind of protocols, if the execution takes more than five minutes it is possible to suppose, considering the complexity of the execution of the original protocol, that this execution is infinite. After running once, and having the first results, we saw that some files stopped after two minutes. So we made another test with TimeOut equals to `300` seconds.

IV. Our work

As stated in the chapter above, our work consisted in finding a way to verify the security of existing protocols using the ProVerif tool, on all the bracketing possibilities of the tuples inside a protocol specification.
As it is not certain that the multiplicity of possibilities will have any effect on the security of the protocol, the second phase of our work heavily depends on the results given by ProVerif. Indeed, we will have to check if the new outcomes given match the original one and if not, we will need to analyse them in order to better understand what exactly changed.

A. Overview

Let's explain the process that we followed in order to check all the possibilities for a given protocol.
In the rest of this document, a "file" is a file in '.pv' format, describing a protocol, written in ProVerif.

Goals: One goal which could be interesting would be, for a given protocol file, to generate all the combinations of bracketing for a tuple. So, all the combinations to "understand" the protocol will be checked.
For instance, sending `((a,b),c)` while the recipient is waiting for `(e,f)` could be a mean to send information not checked by a protagonist. This issue is not found if each n-tuple is considered as a unique structure for a given integer n.
To allow all the possibilities, for a tuple, it would be necessary to generate all the 2-tuples.
Example: *A file containing the following tuples :* (a_1, a_2, a_3, a_4) *and* (b_1, b_2, b_3) *should generate the following files :*

1) $(a_1, (a_2, (a_3, a_4)))$, $(b_1, (b_2, b_3))$
2) $(a_1, ((a_2, a_3), a_4))$, $(b_1, (b_2, b_3))$
3) $((a_1, a_2), (a_3, a_4))$, $(b_1, (b_2, b_3))$
4) $((a_1, (a_2, a_3), a_4))$, $(b_1, (b_2, b_3))$
5) $(((a_1, a_2), a_3), a_4)$, $(b_1, (b_2, b_3))$
6) $(a_1, (a_2, (a_3, a_4)))$, $((b_1, b_2), b_3)$
7) $(a_1, ((a_2, a_3), a_4))$, $((b_1, b_2), b_3)$
8) $((a_1, a_2), (a_3, a_4))$, $((b_1, b_2), b_3)$
9) $((a_1, (a_2, a_3), a_4))$, $((b_1, b_2), b_3)$
10) $(((a_1, a_2), a_3), a_4)$, $((b_1, b_2), b_3)$

How many tuples are there?
How many files should be generated?

Choices and Strategy:
About the Catalan number: The number of generated tuples for a n-tuple is given by the Catalan number.
For an integer n, the $n-th$ Catalan number is defined by :

$$n \in N, C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k}$$

So, for a given n-tuple, the number of existing combinations is growing exponentially (if the factorial function is approximated like exponential, where the exponent is a polynomial function).

Tuples generated: Let's study a given n-tuple t.
$t = \{a_i\}_{i \in [\![0,n-1]\!]}$.
For $k \in [\![0, n-1]\!]$, let's call $left_k$ (respectively $right_k$) the left part (respectively right part) of t, *i.e.* $left_k = \{a_i\}_{i \in [\![0,k-1]\!]}$ (respectively $right_k = \{a_i\}_{i \in [\![k,n-1]\!]}$).
Now, let's call U_{lk} (respectively U_{rk}) the set of all the existing combinations for $left_k$ (respectively $right_k$).
So, all the combinations for this cut are given by the product set $U_{lk} \times U_{rk}$.

So, for this tuple t, all the combinations are given by :

$$\left\{ \bigcup_{k=1}^{n-1} U_{lk} \times U_{rk} \right\}$$

Now, U_{lk} and U_{rk} are in fact the same functions, which we can called $cut(E)$, where E is either the left side or the right side of t. So, the combinations are given by $cut(\{a_i\}_{i \in [\![0,k-1]\!]}) \times cut(\{a_i\}_{i \in, n-1]\!]})$ = $cut(left_k) \times cut(right_k)$.

In fact, this function cut is a recursive-function, using this cutting with k from 0 to $n-1$.
The length of the tuple defines a base case.

$$cut(t) = \begin{cases} \emptyset & if\ card(t) = 0 \\ A = a_0 & if\ card(t) = 1 \\ (A) = (a_0, a_1) & if\ card(t) = 2 \\ \bigcup_{k=1}^{n-1} cut(left_k) \times cut(right_k) & else \end{cases}$$

Once the combinations for a given tuple are found, we have to generate the new files for this given protocol specification.

Files generated: Which files (and how many) are we going to generate?

Choice about bracketing
Given a `out(a,b,c)` message, let us define one of the output that our algorithms can generated as follows : `out((a,b),c)`. Now, on the other part of the protocol there must be a line matching this output. This line corresponds to the receiving part. This line of code starts with the `let/in` keywords. It is clear that those two lines must be generated using the same bracketing format. Indeed, there is no reason to bracket a message in one way and the same message later on in another. In fact, as the person cannot know the content of the message, we can suppose that for one given format, the generation for all the tuples of this format will be the same. And in fact, this is more consistent to use this method because not only there is no reason for a person to decide to proceed with another choice of bracketing but the

algorithm is implemented for all the protagonists being part of the protocol. That is why for a given format the generation is deterministic and cannot propose several possibilities. Moreover if the format used to exchange messages between two machines A and B is different, they will simply not manage to exchange messages and it fails the first purpose of the protocol which is allowing the exchange of messages between machines. Thus it is completely mandatory to generate those lines using the exact same format.

Besides, it is possible for a person to send `out(a,b,c)` like `out((a,b),c)`. But in another exchange send a longer or shorter message with a different format. For instance the protagonists can decide to send something like `(d,((e,f),g))` for this original tuple : `(d,e,f,g)`.

Number of files

That is why we made the choice to regroup the tuples by size and to bracket all group of tuple in the same way. For instance, all the 3-tuples could be generated like `(a,(b,c))` and all the 4-tuples like `((d,e),(f,g))`.

The number of combinations for a kind of tuple (identified by its size) is given by the Catalan number. So, to check all the possibilities, if F_n is the set of the combinations for the n-tuples (so $card(F_n) = C_n$), if n_i is the set of the existing tuples sizes, the combinations of format is given by the product set $F = F_0 \times (F_1 \times (F_2 ...))$.

Let's call S the set of the sizes of the tuples existing in a given file. For instance $S = \{2, 5, 6\}$ if this file contains (and contains only) some 2-tuples, 5-tuples, and 6-tuples (the number of tuples in each category, *i.e.* of each size, is not important). So the number of files is :

$$N_f = card(F) = \prod_{i \in S} card(F_i) = \prod_{i \in S} C_i$$

In the rest of the document, $N_f = card(F)$ is the number of files generated for a given file.

B. Implementation

Vocabulary:

- Let's pay a more detailed attention to the definition of what really is a **tuple** inside the description of a protocol. Of course, we are only studying here the tuples which are exchanged between two protagonists (for instance `(session_id, size, (message, host_B), host_A)` is a real tuple because it defines information which is sent to anther person). However, `sign(message, private_key_A)` or `out(host_A,message)` are functions and not "packages of information" and that is why we don't have to take this kind of tuples into consideration. So let's only call "tuple" the "information-tuple".
- A **combination** is a possible combination for all the tuples of a given size. For instance, `(a,(b,c),d)` is a combination for all the 4-tuples.
- **Combinations** are a "combination of combinations", that is a "set of combination/format for a given size". For instance, "*Let's bracket all the* 3*-tuples like* `(a,(b,c))` *and all the* 4*-tuples like* `((a,b),(c,d))`" is a possible combination for a file which would contain (and only contain) 3-tuples and 4-tuples (and possibly 2-tuples and/or 1-tuples because in our definition of the function cut, the number of possibilities for such tuples is 1). It is important to notice that the number of all the possible combinations for a given file is N_f (The 0-tuples will not be taken into consideration).

Tools: We have used several tools and programming languages to process.

- First of all, **ProVerif** is the software designed to check the safety of a given protocol. It has its own language.
- **Antlr** is the software used to write the grammar and then to generate the parser/lexer [12].
- We used **Java** to set up Antlr and implement a "Visitor", actions done on each syntactic rules after the parsing of a file. We used this language to handle the combinations and create all the files.
- We used **C** programming language (and **gcc** compiler) to execute ProVerif on each file and to compare the result with the one of the original file.

Description of the algorithm: Now, let's suppose that we have the program parsing a given file in the ProVerif language.

Strategy: So, for a given file, we have to generate all the combinations, and, for each combination, generate a new file. In order to know the combinations, we need to know all the tuples which exist in this file, then generate , for each tuple, its possible bracketing, which allows us to generate all the "combinations of bracketing" grouped by size. Finally, for each of these combinations we can generate the right file.

Parsing: Before creating the files, we need to parse the original file. While parsing it, we keep in memory each tuple that we come across.

Firstly we wanted to insert in the grammar the operations to execute, like keep in memory the tuple once the rule is matched, but the parser creation tool we used (Antlr) does not allow this kind of operations. So we have implemented a "visitor" which executes for each rule some operations. Once the corresponding rules of the grammar are matched, a structure of `Tuple` is created.

Remark : At first, we thought that the problem of tuple matching would be easy to implement and we thought about managing those operations by creating a simple parser that would parse the files character by character and match the tuples. But, obviously, this was not as simple as that, as for instance, function calls and function definitions are also matching the format of a tuple and so it was mandatory for

us to use a syntactic parser such as Antlr, in Java.

Coming back to the creation of a tuple, in this structure `Tuple tuple` we generate all the existing combinations for this tuple as a field of the structure. The combinations for a given tuple are generated as we have explained in the formal definition (for each possible cut of the n-tuple in left-member and right-member, we do the product set of the possibilities for the two members).

Now, we have stored in a structure `Tuples tuples` all the `tuple` (which have in its fields its size and its combinations). Thanks to the formula written above, we can compute the number of files N_f to generate.

Creating a file: As we have said, all the tuples of a given size will be bracketed in the same way. So, we have to list all the sizes of tuples existing in the file, and then generate each combinations, *i.e.* for each combination for a given size, generate all the combinations for the other sizes.
Example: *If there are (only) 2-tuples 3-tuples and 4-tuples in a given file, the combinations are* (i, j, k) *format, where* i *is the* i*-th bracketing for the 2-tuples,* j *the* j*-th bracketing for the 3-tuples,* k *the* k*-th bracketing for the 4-tuples. So the combinations are* $(0, 0, 0)$*,* $(0, 0, 1)$*,* $(0, 0, 2)$*,* $(0, 0, 3)$*,* $(0, 0, 4)$*,* $(0, 1, 0)$*,* $(0, 1, 1)$*,* $(0, 1, 2)$*,* $(0, 1, 3)$*,* $(0, 1, 4)$*.*

Then, with the number of the file, we can have the right combinations (in the example above $(0, 1, 2)$ for instance).
We then have to write, for each file, its new version with the matching combinations. In order to do that, we copy the original file and, when the parser comes across a tuple, it is replaced by the corresponding combination. In fact, knowing the number of file that are being written (that is the combinations number) we can have the number of each combination (In the example above, we know that the file's 7-th file, number 6, will be generated using the combinations $(0, 1, 1)$.). Finally, we write the k-th combination of the tuple instead of the original text.

How to find the number of combinations with only the number of the file being created? To find the right combination for each tuple knowing only the file number `i`, we use counter-like structure, where the base is different for each digit. Each digit represents a size, and contains the k-th combination for all the tuples of this size. Indeed, here, each cell is a number, *a priori* on a different base. However, only `i`, the number of the file, is known. The maximum of the cell is the number of possible bracketing for the tuples of the size corresponding to this range. For instance, if the 3-rd cell of our structure corresponds to the tuples of size 4, the number in this cell could be between 0 and 5. And so we have to find the combination (set of) combinations/bracketting corresponding.
Example : *To illustrate this issue, it is possible to draw a parallel with a time-counter. Let's consider 19200s, and we have to find from this number (which corresponds to the number of the file in our case) the amount of days, hours, minutes, and seconds. There are 60 seconds in a minute, 60 minutes in an hour, but 24 hour in a day. The solution to find is* (X_d, X_h, X_m, X_s)*, with* $19200seconds = X_d days + X_h hours + X_m minutes + X_s seconds$*. How would you do to find these numbers from the original number (*19200*) only?*

Implementation : There are two ways to proceed. Either by several euclidean divisions and using the integer part and the rest, recursively, the base changing (which is the divisor) to each function call (which the value is the number of possible combinations for a tuple of the current size).
Or, the other way, *a priori* easier to implement, is to increment the counter for each loop step.

Writing: Now, for each step of the loop on the number of files, we know for each size which bracketing we have to choose. We have implemented a structure that contains the lines where there is a tuple (and containing this tuple). During the creation of a tuple, the line is noticed to this structure. So that when writing the file, we can copy all the lines and when a line should be modified, we replace the tuple by its current combination given by the number of the file being created.

Details: For further details about the practical implementation, please refer to the Annex B, where implementation choices, details of the strategy, and possible optimisations are detailed.

Running ProVerif: Now, we have all the possibilities of tuple-bracketing. We have automated the execution of ProVerif on these files and the comparison with the output of the original file.
We did that in C language. The single parameter is a directory. All the sub-directories will be parsed, and a sub-directory containing files should be a protocol, where the original is the number 0 and the other files are other possibilities. This format for the name of the files is the output of our program written in Java which creates all the files. Now, for each protocol, ProVerif is run on the original file, and then for each generated file, the output is compared to that of the original, and a summary of the results is given in a `txt` file in this directory.

V. RESULTS

A. An interesting result on the Yahalom protocol

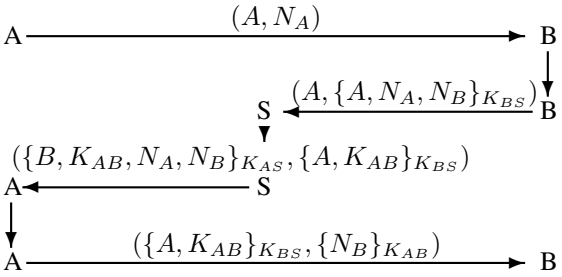
Now let us study all the files marked as "different", *i.e.* the output of the file is not the same as the one of the original file. As we have said, the output is one of the four possible result which is not necessarily a ProVerif output (like TimeOut).

The original file (SimplerYahalom-unid.pv, available in Annex A) is marked as "safe", but traces are found in the 2^{nd} file generated (SimplerYahalom-unid_2.pv, also available in Annex A).
Let's describe the original protocol and then its attack on the bracketing.

Description of the protocol: Here is the description of the Yahalom Protocol (Fig. 9).

- S is here a trusted sever, "arbitror".
- N_X is a nonce generated by X.
- K_{XY} is a symmetric key shared between and (only known by) X and Y.
- Here, K_{AB} is generated by S. It will be the session key between A and B.

Fig. 9. Description of the Yahalom Protocol



<u>*Nota Bene :*</u> The file describing the original Yahalom protocol is, in the folder "`examples/pitype/secr-auth`" of the software ProVerif is `Yahalom.pv`. However, for each protocol, there are many variants of the protocol, with improvements or other changes. For instance, for Yahalom in "`secr-auth`" (which is the folder where the property check id "secret and authentication"), there are the following files about the Yahalom protocol :

- `Yahalom.pv`,
- `Yahalom-block-cipher.pv`,
- `Yahalom-Paulson.pv`,
- `Yahalom-proba-enc.pv`,
- `SimplerYahalom.pv`,
- `SimplerYahalom-unid.pv`.

Let's compare `Yahalom.pv`, describing the original protocol, and `SimplerYahalom-unid.pv`, the variant we have described above. That is the same protocol, so the structure of the conversation is the same.

`Yahalom.pv` :

$$\begin{array}{ll} A \to B & : A, N_A \\ B \to S & : B, \{A, N_A, N_B\}_{K_{BS}} \\ S \to A & : \{B; K_{AB}; Na, N_B\}_{K_{AS}}, \{A, K_{AB}\}_{K_{BS}} \\ A \to B & : \{A, K_{AB}\}_{K_{BS}}, \{N_B\}_{K_{AB}} \end{array}$$

`SimplerYahalom-unid.pv`
and **`SimplerYahalom.pv`** :

$$\begin{array}{ll} A \to B & : A, N_A \\ B \to S & : B, N_B, \{A, N_A\}_{K_{BS}} \\ S \to A & : N_B, \{B, K_{AB}, N_A\}_{K_{AS}}, \{A, K_{AB}, N_B\}_{K_{BS}} \\ A \to B & : \{A, K_{AB}, N_B\}_{K_{BS}}, \{N_B\}_{K_{AB}} \end{array}$$

As can be seen, the only difference in the protocol between these two variants is the way to send N_B. The queries checked are here the same, but in other folders ("`non-interf`" for instance), other properties could be checked.
The properties checked are the same. For instance,

```
free secretA, secretB: bitstring [private].
query   attacker(secretA);
        attacker(secretB).
```

That means that none of the secrets should be known by the attacker.

Attack: One of the file created ("SimplerYahalom-unid_2.pv" in the Annex A) is shown as having an attack. Let's study the trace.
First, we have these messages (Fig. 10). With the help of the protocol, we have to find whose message it is.

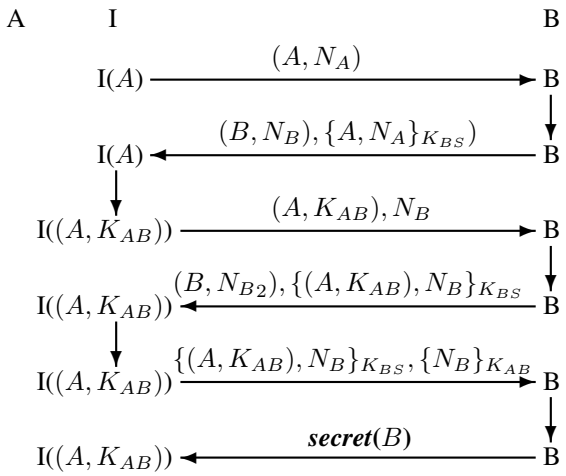
Fig. 10. Messages of attack of the Yahalom Protocol (Trace)

in	(A, N_A)	(1)
out	$((B, N_B), \{A, N_A\}_{K_{BS}})$	(2)
in	$((A, K_{AB}), N_B)$	(3)
out	$((B, N_{B2}), \{(A, K_{AB}), N_B\}_{K_{BS}})$	(4)
in	$(\{(A, K_{AB}), N_B\}_{K_{BS}}, \{N_B\}_{K_{AB}})$	(5)
out	***secret*(B)**	(6)

Now, we can suppose that the conversations are as described below (Fig. 11).

In the first part of the conversation ((1) and (2)), I is impersonating A. Then, I is impersonating a fictive

Fig. 11. Description of attack of the Yahalom Protocol



protagonist (A, K_{AB}) (messages (3), (4), (5) and (6)). So, at the end of the conversation, I knows information that B sends to I, that I knows thanks to K_{AB}.
You can note that here the host A does not intervene in the process, which is quite amazing.

Nota Bene: *After the analyse of the file (for instance with the online tool available here), there are several traces. However, the other traces are similar to the one presented above.*

B. Listing of all the results

Throughout the execution of our scripts on the multiple examples that we were given, most of them ended up giving no result at all, the complete list of the protocol tested is available in the table below. It is worth noticing that are included inside the files that we tested few variations of the original protocols. As explained earlier, it is possible for a protocol to be tested on multiple security properties, and sometimes it is needed to build a slightly different version of the protocol to test them. Moreover, it can be interesting to try and check if a few modifications such as the correction of an attack or a different kind of encryption would make any difference in the output given by ProVerif, that is the reasons behind the multiples variations of a single protocol.

Now, on some protocols the results given by ProVerif were different from the original one. For some, it is due to a time out of ProVerif : as explained before, the execution was too long for us to keep going and is a sign of a problem on the ProVerif end. For some others the result is different, whether they are safe, or not, and even not proven. In the following table are listed the complete list of results. This table puts a special emphasis on the impact of the bracketing on the execution of ProVerif. Thus, it is possible to see the percentage of generated files for which the result changed and amongst those the actual reason behind it, still expressed in percentage (Figure V-B).

It is worth noticing that on the multiple tested protocols only a few of them are knowing significant change concerning their ProVerif outputs. Indeed, the three protocols in question are Otway-Rees [11], Yahalom [5] and Needham-Schroeder [7]. Most of the generations ended up timing out to the ProVerif execution (reminder: the time out used is five minutes long). In this table it is also clear that the only different result in outputs leading to the discovery of an attack is a stated above, on the Yahalom protocol.

Remark : For all the other protocols that do not have any different results, the execution of the ProVerif verification took a lot more time compared to the original version of the protocol. As an example, the complete execution on a single machine with a five-minutes time out took about seventy hours to finish and execute the 2210 generated files whereas the execution only takes a few seconds to end on the 72 original files.

TABLE I
TABLE OF RESULTS ON ALL THE TESTED PROTOCOLS

Protocol	**Variations**	**Generations**	**Results (%)**			
			Different result	Timed out	No proof	Different
Otway-Rees	9	1970	43	80	20	0
Yahalom	6	64	48	90	6	4
Needham-Schroeder	15	76	50	100	0	0
Diffie-Hellman	2	0	0	0	0	0
Denning-Sacco	6	28	0	0	0	0
Woo-Lam	6	10	0	0	0	0
Skeme	3	30	0	0	0	0
ssh-transport	1	10	0	0	0	0
basic	6	0	0	0	0	0
dh-fs	1	2	0	0	0	0
epassportUK	4	8	0	0	0	0
handshake	2	0	0	0	0	0
macs	1	0	0	0	0	0
private_authentication	3	6	0	0	0	0
proba-pk	1	0	0	0	0	0
vote	2	0	0	0	0	0
wmf	4	6	0	0	0	0

VI. Conclusion

Throughout this paper, we firstly described the general situation and explained the problem that is encountered in the formal verification processes. Then, with the study of the ProVerif verification tool, we explained precisely what is a tuple and how it is defined as well as the important security properties that are verified. Next we explained in details our contribution to the matter and with it presented our results on a set of protocols.

This results are really interesting. At first, the most important part of it is the discovery of a new attack on a version of the Yahalom protocol. But every other result has its importance as well as it is a proof that the formal verification of the security of protocols is a very delicate matter. One could assume at first, as said through this paper, that the modification of the bracketing of a tuple, should not have any impact on the verification of it and if so, this impact should be minimum. Yet, it is shown by the conducted experiments that it has a strong impact. This impact touches not only the output given by ProVerif but, more generally, the time that ProVerif needs to give this output.

Future works: In order to complete and push further this work, it is possible to improve our programs to reuse them in a slightly more appropriate way. Indeed, the verification of the bracketing combination of a protocol could be integrated directly into the ProVerif compiler written in OCaml, as an option when starting the verification.

Moreover, as it has been explained in sections above, our programs do not take into account the possible position changes of elements inside a given tuple. With the Needham-Schroeder-Lowe protocol, for instance, we saw that this position change could lead to an attack in combination with a bad bracketing. It would then be a great addition to our already functioning program.

Finally, it would be interesting to use our program in order to verify some other protocols, like protocols that might be more complex and that could potentially hide attacks behind bracketing, for instance. As the experiments show, it is most likely that a lot of protocol's verification are impacted in some way by the change of bracketing.

VII. Acknowledgements

We would like to thank our two supervisors during this entire project : Mr. Vincent Cheval and Ms. Véronique Cortier who both gave a lot of their time in order to encourage us to go further into the discovery of what was a completely new subject to us. They helped us apprehend the subject, its context and the tool associated with it: ProVerif and its harsh looking interfaces. It has not been as straightforward as we could have imagine before really diving into it. It is clear that without the help of our two supervisors and our regular meetings, it would have been really hard for us to come up with the work presented in the paper. Moreover, they introduced us to the research environment that we were also unfamiliar with, especially the method behind the writing of a report, and the general way of approaching a large problem. Although any error in this paper is our own, they helped us review it and gave us valuable insight in the writing of it.

REFERENCES

[1] D. Bassin, C. Cremers, J. Dreier, and R. Sasse. Symbolically analyzing security protocols using tamarin. 2017.

[2] B. Blanchet. A computationally sound automatic prover for cryptographic protocols. 2005.

[3] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[4] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. Proverif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial, 2018.

[5] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *Proceedings of the Royal Society of London*, 1989.

[6] A. Freier, P. Karlton, and P. Kocher. The secure sockets layer (SSL).

[7] R. Stockton Gaines, Roger M. Needham, and Michael D. Schroeder. Using encryption for authentication in large networks of computers.

[8] Jeffrey I. Schiller Jennifer G. Steiner, Clifford Neuman. Kerberos: An authentifcation service for open network systems.

[9] Gavin Lowe. An attack on the needham-schroeder public key authentication protocol. *Information Processing Letters*, 1995.

[10] Durgin Lincoln Mitchell, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. 1999.

[11] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM Operating System Review*, 1987.

[12] Terence Parr. *The Definitive ANTLR Reference, Building Domain-Specific Languages*. Pragmatic Bookshelf, 2nd edition, 2013. https://doc.lagout.org/programmation/Pragmatic%20Programmers/The%20Definitive%20ANTLR%20Reference.pdf.

[13] E. Rescorla T. Dierks. The transport layer security (TLS) protocol, version 1.2. 2008.

[14] L. Vigan. Automated security protocol analysis with the avispa tool, 2007.

VIII. Annexes

A. Annex A : SimplerYahalom-unid pv files

Here above are two files describing by the same way the Yahalom protocol.

The file marked as _0 is the original file (without changes). The file marked as _2 is one of the generated file where there is the attack we have talked about. We can notice that indeed the tuples are bracketted as described. Here (in the _2 file), the 2-tuples are obviously bracketted like $(a_1, a_2$, the 3-tuples are bracketted like $((a_1, a_2), a_3)$.

Maybe that is clearer to see the differences above between these files in Fig. 12.

files/SimplerYahalom–unid_0.pv

```
free c: channel.

type key.
type host.
type nonce.

fun nonce_to_bitstring(nonce): bitstring
    [data,typeConverter].

(* Shared key encryption *)

fun encrypt(bitstring,key): bitstring.
reduc forall x: bitstring, y: key;
    decrypt(encrypt(x,y),y) = x.

(* Secrecy assumptions *)

not attacker(new Kas).
not attacker(new Kbs).

(* 2 honest host names A and B *)

free A, B: host.

(* the table host names/keys
   The key table consists of pairs
   (host, key shared between the host and
       the server) *)
table keys(host, key).

(* Queries *)

free secretA, secretB: bitstring [private
    ].
query attacker(secretA);
      attacker(secretB).

event endAparam(host, host).
event endBparam(host, host).
event beginAparam(host, host).
event beginBparam(host, host).
event endBkey(host, host, nonce, key).
event beginBkey(host, host, nonce, key).

query x: host, y: host; inj-event(
    endAparam(x, y)) ==> inj-event(
    beginAparam(x, y)).
query x: host, y: host; inj-event(
    endBparam(x, y)) ==> inj-event(
    beginBparam(x, y)).
query x: host, y: host, z: nonce, t: key;
     inj-event(endBkey(x,y,z,t)) ==> inj-
    event(beginBkey(x,y,z,t)).

(* Role of the initiator with identity xA
     and key kas shared with S *)

let processInitiator(xA: host, kas: key)
    =
  new Na: nonce;
  out(c, (xA, Na));
  in(c, (nb: nonce, m1: bitstring, m2:
      bitstring));
        let (b: host, kab: key, na2:
            nonce) = decrypt(m1, kas) in
  event beginBparam(b, xA);
  event beginBkey(b, xA, nb, kab);
        if na2 = Na then
        out(c, (m2, encrypt(
            nonce_to_bitstring(nb), kab)))
            ;
  (* OK protocol finished
     If the interlocutor is honest,
        execute the events endAparam
           and send a test message to
              check that the key kab is
              secret *)
  if b = A || b = B then
  event endAparam(xA, b);
  out(c, encrypt(secretA, kab)).

(* Role of the responder with identity xB
     and key kbs shared with S *)

let processResponder(xB: host, kbs: key)
    =
  in(c, (a: host, na: nonce));
  event beginAparam(a, xB);
        new Nb: nonce;
  out(c, (xB, Nb, encrypt((a, na), kbs)))
      ;
  in(c, (m3: bitstring, m4: bitstring));
        let (=a, kab: key, =Nb) = decrypt
            (m3, kbs) in
        if nonce_to_bitstring(Nb) =
```

```
            decrypt(m4, kab) then
  (* OK protocol finished
            If the interlocutor is honest,
                execute the events
               endBparam
            and endBkey, and send a test
                message to check that the
                key kab
     is secret *)
  if a = A || a = B then
  event endBparam(xB, a);
  event endBkey(xB, a, Nb, kab);
  out(c, encrypt(secretB, kab)).

(* Server *)

let processS =
  in(c, (b: host, nb: nonce, m5:
      bitstring));
  get keys(=b, kbs2) in (* get the key of
      b from the key table *)
        let (a: host, na: nonce) =
            decrypt(m5,kbs2) in
  get keys(=a, kas2) in (* get the key of
      a from the key table *)
        new kab: key;
  out(c, (nb, encrypt((b, kab, na), kas2)
      , encrypt((a, kab, nb), kbs2))).

(* Key registration *)

let processK =
        in(c, (h: host, k: key));
        if h <> A && h <> B then insert
            keys(h,k).

(* Start process *)

process
  new Kas: key; new Kbs: key;
  insert keys(A, Kas);
  insert keys(B, Kbs);
  (
          (* Launch an unbounded number
              of sessions of the initiator
              *)
          (!processInitiator(A, Kas)) |
          (* Launch an unbounded number
              of sessions of the responder
              *)
          (!processResponder(B, Kbs)) |
          (* Launch an unbounded number
              of sessions of the server *)
          (!processS) |
          (* Key registration process *)
    (!processK)
          )
```

Here the file generated described in the introduction of the Annex A.

files/SimplerYahalom–unid_2.pv

```
free c: channel.

type key.
type host.
type nonce.

fun nonce_to_bitstring(nonce): bitstring
    [data,typeConverter].

(* Shared key encryption *)

fun encrypt(bitstring,key): bitstring.
reduc forall x: bitstring, y: key;
    decrypt(encrypt(x,y),y) = x.

(* Secrecy assumptions *)

not attacker(new Kas).
not attacker(new Kbs).

(* 2 honest host names A and B *)

free A, B: host.

(* the table host names/keys
   The key table consists of pairs
   (host, key shared between the host and
       the server) *)
table keys(host, key).

(* Queries *)

free secretA, secretB: bitstring [private
    ].
query attacker(secretA);
      attacker(secretB).

event endAparam(host, host).
event endBparam(host, host).
event beginAparam(host, host).
event beginBparam(host, host).
event endBkey(host, host, nonce, key).
event beginBkey(host, host, nonce, key).

query x: host, y: host; inj-event(
    endAparam(x, y)) ==> inj-event(
    beginAparam(x, y)).
query x: host, y: host; inj-event(
    endBparam(x, y)) ==> inj-event(
    beginBparam(x, y)).
```

```
query x: host, y: host, z: nonce, t: key;
    inj-event(endBkey(x,y,z,t)) ==> inj-
    event(beginBkey(x,y,z,t)).

(* Role of the initiator with identity xA
    and key kas shared with S *)

let processInitiator(xA: host, kas: key)
    =
  new Na: nonce;
  out(c, (xA, Na));
  in(c, ((nb: nonce, m1: bitstring), m2:
      bitstring));
        let ((b: host, kab: key), na2:
            nonce) = decrypt(m1, kas) in
  event beginBparam(b, xA);
  event beginBkey(b, xA, nb, kab);
        if na2 = Na then
        out(c, (m2, encrypt(
            nonce_to_bitstring(nb), kab)))
            ;
  (* OK protocol finished
     If the interlocutor is honest,
         execute the events endAparam
            and send a test message to
                check that the key kab is
                secret *)
  if b = A || b = B then
  event endAparam(xA, b);
  out(c, encrypt(secretA, kab)).

(* Role of the responder with identity xB
    and key kbs shared with S *)

let processResponder(xB: host, kbs: key)
    =
  in(c, (a: host, na: nonce));
  event beginAparam(a, xB);
        new Nb: nonce;
  out(c, ((xB, Nb), (encrypt((a, na), kbs
      ))));
  in(c, (m3: bitstring, m4: bitstring));
        let ((=a, kab: key), =Nb) =
            decrypt(m3, kbs) in
        if nonce_to_bitstring(Nb) =
            decrypt(m4, kab) then
  (* OK protocol finished
           If the interlocutor is honest,
                execute the events
                endBparam
              and endBkey, and send a test
                message to check that the
                key kab
     is secret *)
  if a = A || a = B then
  event endBparam(xB, a);
  event endBkey(xB, a, Nb, kab);
  out(c, encrypt(secretB, kab)).

(* Server *)

let processS =
  in(c, ((b: host, nb: nonce), m5:
      bitstring));
  get keys(=b, kbs2) in (* get the key of
      b from the key table *)
        let (a: host, na: nonce) =
            decrypt(m5,kbs2) in
  get keys(=a, kas2) in (* get the key of
      a from the key table *)
        new kab: key;
  out(c, ((nb, encrypt(((b, kab), na),
      kas2)), (encrypt(((a, kab), nb),
      kbs2)))).

(* Key registration *)

let processK =
        in(c, (h: host, k: key));
        if h <> A && h <> B then insert
            keys(h,k).

(* Start process *)

process
  new Kas: key; new Kbs: key;
  insert keys(A, Kas);
  insert keys(B, Kbs);
  (
          (* Launch an unbounded number
              of sessions of the initiator
              *)
          (!processInitiator(A, Kas)) |
          (* Launch an unbounded number
              of sessions of the responder
              *)
          (!processResponder(B, Kbs)) |
          (* Launch an unbounded number
              of sessions of the server *)
          (!processS) |
          (* Key registration process *)
    (!processK)
        )
```

Fig. 12. Differences between the two following files

```
$ ls
SimplerYahalom-unid_0.pv  SimplerYahalom-unid_1.pv  SimplerYahalom-unid_2.pv
$ diff SimplerYahalom-unid_0.pv SimplerYahalom-unid_2.pv
94,95c94,95
<       in(c, (nb: nonce, m1: bitstring, m2: bitstring));
<         let (b: host, kab: key, na2: nonce) = decrypt(m1, kas) in
---
>       in(c, ((nb: nonce, m1: bitstring), m2: bitstring));
>         let ((b: host, kab: key), na2: nonce) = decrypt(m1, kas) in
113c113
<       out(c, (xB, Nb, encrypt((a, na), kbs)));
---
>       out(c, ((xB, Nb), (encrypt((a, na), kbs))));
115c115
<         let (=a, kab: key, =Nb) = decrypt(m3, kbs) in
---
>         let ((=a, kab: key), =Nb) = decrypt(m3, kbs) in
129c129
<       in(c, (b: host, nb: nonce, m5: bitstring));
---
>       in(c, ((b: host, nb: nonce), m5: bitstring));
134c134
<       out(c, (nb, encrypt((b, kab, na), kas2), encrypt((a, kab, nb), kbs2))).
---
>       out(c, ((nb, encrypt(((b, kab), na), kas2)), (encrypt(((a, kab), nb), kbs2)))).
$
```

B. Annex B : Practical implementation

Implementation of the parsing: Let's detail how we proceed in order to implement the parsing of an original file and the creation of the files which should be generated.
As we have said, we implement this part in `Java` ; so the following commands and the specific vocabulary are relative to the `Java` language.

Aim of the classes: Let's present the classes we have created.

- A **`Tuple`** is an `ArrayList` in which the elements are the elements of the Tuple. Built with its context, it contains its original form and will contain all its combinations.
- **`Tuples`** is an `ArrayList`, global list of the tuples present in a given file. For a given file (original file, not a generated file), there is only one instance of this class.
- **`Combinations`** is a final class, is built once the file is parsed and contains the list of the tuples. It is not designed to be changed once built. It generates all the combinations for each tuple, and then indicates how many files will need to be generated.
- **`CombinationsHandler`** knows the instance (because unique) of combinations. It aims to handle the creation of each child-file.

Sequence of events:

- Parsing of the original file
  - If a `Tuple tuple` is checked :
    - An object `Tuple tuple` containing this one is created
      - All combinations are generated as an attribute-field of this object.

        Implementation : The new file is not generated here because Antlr does not handle perfectly the insertion. And would not allow us to have a good amount of control over the creation of the files.

    - `tuple` is added to the list `Tuples tuples`, the list of all the tuples of the given file.
    - The modification of a line is indicated in a `HashMap`, it notifies for each line which tuple is linked to it. As there may be several tuples for a given line, the key of the `HashMap` is the line number, but the value is an `ArrayList` of Tuple. So the `HashMap` is defined like this : `HashMap<int,ArrayList<Tuple>>`.

    Implementation : Indeed, when the file is being created, it allows usto go directly to the lines which should be modified. Thus it prevents us from doing more replace operations than there are tuples in the file.

    What's more, there may be several tuples in a line and a `HashMap` helps to optimise this management.

- The number of file which need to be created is calculated.

  Implementation : `sizes` is the list of sizes of tuples present in the original file.

  Example : *`sizes=3,6,7` means that the file contains and contains only tuples of size 3, 6, and 7.*

- Let us loop through this number of files.
  For a file number `i` :
  - The file number `i` is created.
  - The right set of combinations. To find the right combination for each tuple knowing only the file number `i`, we use counter-like structure, where the base is different for each digit. Each digit represents a size, and contains the k-th combination pour all the tuples of this size. It is also noteworthy that is consistent with the formula (i) which corresponds to the maximum of the counter.

    Implementation : `indexOfCombinations` provides information for each size of which combination (combination number) must be used. Thus the size of `indexOfCombinations` is `sizes.size()`. Furthermore, `therefore.get(k)` provides information about the combination of tuples of k-element, corresponding to a size equals to `sizes.get(k)`.
    In practice, we need to move up the information string the other way around. From a tuple, we would like to know its combination. So we have to know its index in `indexOfCombinations`, which we can obtain, for a given tuple, by looking into `sizes.indexOf(tuple.size())`.
    Finally, it is also worthy of note that for a tuple of size k, its cell in `indexOfCombinations` ranges from 0 to the number of possible combinations for a tuple of size k.

    Example : *Continuing the previous example, if `indexOfCombinations=1,3,0`, all tuples of size 3 will be bracketted depending on the 2nd bracketting (number 1), those of size 6 with the 4th, and those of size 7 with the 1st.*

    - Each tuple is parsed and its combinations for the file number i is stocked.
  - The file number i is written by reading the original file line by line.
    - If the line number i must undergo a change :
      - For each tuple linked to the line, this one is replaced by the combination corresponding to the file number i.

· The new line is printed.

∗ Else, the line read is printed in the new file

– The numbers of combination are updated for each category of tuple (distributed by size).

Implementation : All combinations of number of combination for each category of tuple must be parsed. As we have said above in the section "Our Work" of the report, we use the counter-like structure. The counter is incremented for each loop step.

Design choices: In addition to the explanations above, maybe the reasons about a few choices should be detailed :

- **Why do we not create the file at the same time we analyse the original file?** There are several reasons why we create the files after the parsing of the original file. Some of these reasons are explained above (need to know the number of files, management of the operations by Antlr analysing a file, ...). But the creation of the data structures, the knowledge of the other objects, the management of the Java imports, are some reasons why we do not replace the tuples 'on the fly'.

- **What about the complexity / optimisation?** Maybe some functions could be a little bit more optimised (for instance, the replacement of several tuples in one line, currently there is one 'replace' by tuple), but this kind of optimisation is not something that will drastically change the complexity of the program that we have today.

- **Are all files able to be parsed?** Currently, our version of the ProVerif grammar contains the main rules (all rules needed to parse the files of the folder "secr-auth", "weak-secr", "choice", "non-interf" from the folder 'examples' in the software ProVerif). The remaining rules can be added, in order to parse other files.

C. Annex C : List of all the protocols tested

Protocol
Denning-Sacco
Diffie-Hellman
Needham-Schroeder
Otway-Rees
Woo-Lam
Yahalom
ssh-transport
basic
dh-fs
epassportUK
handshake
macs
private_authentication
proba-pk
vote
wmf