Lab Report

Automatic Security Protocol Analysis with Tamarin-Prover

Eike Stadtländer

7 February, 2019

Abstract

In the face of attacks against cryptographic protocols both of practical and theoretical kind and with increasing frequency, we are interested in improving the trustworthiness of security proofs. This part of security analyses is prone to human error to a special degree because mistakes in mathematical proofs are often oblique and thus hard to spot. Moreover, the sheer complexity of protocols and their extensions and the consequential length of security proofs are severe bottlenecks. This motivates the use of automated security analysis tools and proof assistants as a solution to this problem. We focus on an evaluation of Tamarin-Prover while working towards an automatic security analysis of the Internet Key Exchange version 2 which is used in IPsec.

Contents

1. Introduction

1.1. Motivation and Relevance

In modern society, cryptographic protocols are ubiquitous though unnoticed: Most people use them on a day-to-day basis without being aware of it. For example, instant messaging applications make extensive use of cryptographic protocols to provide confidentiality, integrity or authenticity. Even more crucial, many government and business operations rely on securing data and the transmission thereof. Hence, it is absolutely essential that we can rely on the security of cryptographic protocols.

Modern cryptography establishes security properties of its primitives and protocols by creating mathematical security proofs. However, experience shows that both theoretical proofs as well as practical implementations cannot always be trusted. For instance, the Heartbleed attack is a security bug in the TLS implementation OpenSSL v1.0.1-v1.0.1f that enabled attackers to obtain sensitive data due to a buffer overflow (MITRE, 2013; Durumeric et al., 2014). And on the theoretical side, Daniel Bleichenbacher was able to find a subtle flaw in the design of the padding of the encoding function in PKCS#1 v1.5 (Bleichenbacher, 1998).

Why is it that theoretical and practical security is so hard to achieve? We argue that the human factor plays a major role in this story. First of all, more often than not there is an unbridgeable gap between the idealized theoretical description and the down-to-earth implementation of the theory. Second, there are already many prominent cases where mathematical proofs contain flaws which are very difficult to detect for humans due to a) the obscurity of the flaws, b) the length and complexity of the proof, c) the time available in the review process (Opfer, 2011; Blum, 2017; Mochizuki, 2018). In practice, protocols like TLS or IPsec offer a large suite of cryptographic primitives and allow for extensions. This creates many different cases and possible interdependencies to be considered when analysing the security. This makes it particularly difficult and time consuming for humans to analyse the security of such protocols. Even more so since extensions, variations and even new kinds of protocols are easier to create than to analyse.

Thus, it seems natural to solve these problems by using an *automated security analysis tool* which either finds security proofs by itself or at least assists at it. While this probably still does not provide perfect security it gives an additional layer of trust in the analysed security properties.

In our lab, we evaluated the suitability and capabilities of such automated security analysis tools with regards to analysing large protocols like IPsec and TLS. More specifically, we made some effort towards an analysis of IPsec using Tamarin-Prover inspired by prior work done for analysing TLS by Cremers et al. (2017), cf. section 1.3.

1.2. Related Work

There already exist automated provers and verification tools along with underlying theories for both the mathematical domain and more specifically the cryptographic domain.

Most notably among the tools working in the general mathematical domain is Coq. This semi-automated prover was applied to the Four-Color Theorem by Gonthier (2008). Tools which are tailored to security analyses include amongst others Tamarin-Prover and EasyCrypt. These tools were evaluated in our lab.

Tamarin-Prover

Tamarin-Prover is a "security protocol verification tool that supports both falsification and unbounded verification in the symbolic model" (Basin et al., 2014). It was initially developed by Schmidt (2012) and Meier (2013) as part of their dissertations at ETH Zurich. Schmidt (2012) laid the foundation of a constraint solving algorithm whose implementation was called Tamarin-Prover. This tool was subsequently applied to analyse several cryptographic protocols. Meier (2013) built upon that work by expanding the verification theory underpinning Tamarin-Prover so that it additionally supports Diffie-Hellman exponentiation, bilinear pairings and multisets (Meier, 2013, pg. 136ff.).

Cremers et al. (2017) applied Tamarin-Prover extensively to analyse the security of TLS1.3 draft 21. They cover analysis of all supported handshake modes and come to the conclusion that the security requirements are met in the symbolic model of Tamarin-Prover.

EasyCrypt

EasyCrypt is a semi-automated proof assistant inspired by Coq and SMT solvers. It was initially developed by the IMDEA Software Institute. Later, Inria and École Polytechnique joined the development and maintenance of EasyCrypt (IMDEA Software Institute, 2009).

In contrast to Tamarin-Prover, EasyCrypt works in a computational model and builds game-based proofs in the same style as described by Lindell and Katz (2014).

We concentrate on the evaluation of Tamarin-Prover. However, an evaluation of EasyCrypt from a very similar perspective as we do here with Tamarin-Prover was done by Nussbaumer and Nüsken (2018). Furthermore, EasyCrypt was used to analyse the security of the TLS1.2 Handshake (Bhargavan, Cédric Fournet, et al., 2014).

F* Programming Language

A last notable mention is the F* programming language which provides verification as a baked-in feature. That means, it is possible to give precise specifications for F* programs and verify the actual programs against their specifications (The F* Team, 2019). There exists a verified reference implementation of TLS1.2 in F* called miTLS (Bhargavan, Cédric Fournet, et al., 2013).

1.3. Our Contribution

As noted before, we were interested in evaluating Tamarin-Prover as a suitable tool for automatic security analysis of cryptographic protocols. As there is far less work on IPsec compared to existing work on TLS, we are particularly interested in analysing the IPsec protocol.

First, we implemented an ephemeral Diffie-Hellman key exchange, an adaptation of a non-ephemeral Diffie-Hellman key exchange by Wong (2017). In this example, two statements were analysed. The first being that the Diffie-Hellman key exchange as implemented in Tamarin-Prover is executable by honest parties in the intended way. The second being that an active attacker can perform a man-in-the-middle attack. We will use this example in section 2.1 to introduce Tamarin-Prover and its language.

Second, we created and discussed building blocks necessary for implementing a cryptographic protocol like IPsec. Although our interest is for the IPsec protocol, we tried to present these building blocks in a general way so that they can in principle be applied to other protocols as well. They include signature schemes, authenticated encryption and Diffie-Hellman exponentiation. The findings of this process are presented in section 2.2

Third, we used our findings and the building blocks to implement the initialization phase of the key exchange used in IPsec. This should be considered as work in progress because the initialization phase alone is only part of IPsec's key exchange. However, our implementation so far was successfully analysed with the following expected findings: First, the honest setup is indeed executable. And second, the security parameter is not secure in the sense that a man-in-the-middle is able to let two parties think they talk to each other but deriving different security parameters. This motivates the authentication phase of this key exchange protocol. Section 3 deals with these results.

Lastly, we discuss our findings in the context of improving the trustworthiness of security proofs and the effectiveness of automated security analysis tooling in section 4.

2. Tamarin-Prover

Tamarin-Prover can verify or falsify lemmata stated in its specification language. As we will argue in section 2.1, lemmata usually represent security properties. Here, verification means that the specified protocol satisfies the respective security property and falsification means that Tamarin-Prover found an attack against the respective security property (The Tamarin Team, 2019).

2.1. An Introductory Example

In this section, we present an annotated implementation and analysis of an ephemeral Diffie-Hellman key exchange protocol using Tamarin-Prover. It is an adaptation of the non-ephemeral Diffie-Hellman key exchange protocol implemented by Wong (2017). The goal of this example is twofold. First, we want to introduce the syntax of the specification language of Tamarin-Prover which is essential for the later sections. And second, we want to illustrate a) how an actual protocol is implemented using this language and b) how the language structures correspond to cryptographic primitives and security properties.

Security Protocol Theories

A *security protocol theory* is a collection of

1. multiset rewriting rules which usually implement a protocol or more precisely the state transitions of a protocol,
2. function symbols which may represent cryptographic primitives such as hash functions or encryptions schemes,
3. equations which define the semantic of said function symbols like the correctness equations connecting decryption and encryption algorithms,
4. lemmata which usually describe the security properties and sanity checks,
5. and other language constructs such as restrictions, built-ins and axioms

most of which will be explained in the following subsections.

The input to Tamarin-Prover is then a security protocol theory written and stored as a single `.spthy` file. Depending on the mode in which it is executed Tamarin-Prover either tries to verify or falsify every lemma specified in the security protocol theory or it may present a semi-automatic interactive mode which can be explored by the user.

Syntax-wise, a security protocol theory is framed by the keyword `theory`, an identifier for the theory and the keywords `begin` and `end`. A skeleton of our soon-to-be example might look as follows:

```
theory DHKE
begin
```

```

builtins: diffie-hellman

// ...
end
```

Using `builtins` one can enable certain features Tamarin-Prover is shipped with. In our case, we need Diffie-Hellman exponentiation. There are also built-ins for symmetric encryption, signatures and bilinear pairings. The rest of our example will unfold from the line marked with the three-dot comment.

Labeled Multiset Rewriting, Traces and Lemmata

Simply put, multiset rewriting is the process of transforming a multiset into another multiset by applying feasible and permitted rewriting rules. For instance, a multiset $X = \{x, x, y, z\}$ might be transformed into or rewritten as the multiset $X' = \{x, y, z, z\}$ if there is a permitted rewriting rule consuming one x and producing one z. There might also be a second rule consuming two ys and producing one z. Since there is only one y contained in X it is not feasible to apply the second rule on X.

Schmidt (2012) and Meier (2013, sec. 7.2, pg. 76ff.) discovered that multiset rewriting can be used to implement the unbounded concurrent execution of cryptographic protocols using rewriting rules which model the state transitions of the protocol instances. As an example, in a client-server setting one might write a rule consuming an idle state of the server and an incoming message of the client to produce a waiting state of the server and an outgoing message to the client. As rules do not only consume and produce states but also messages and other user-defined objects, the entities consumed and produced by a rule are called *premise facts* and *conclusion facts*, respectively.

However, a slight adaptation from classical multiset rewriting was necessary to build a verification theory on top of it. Suppose one wants Tamarin-Prover to verify that a specified protocol is indeed correctly executable for honest parties. Then one could formulate this property in case of a key exchange protocol by saying: "There is a sequence of feasible applications of rules such that–at the end–two parties derive the same keys and correctly assume that each is talking to the other."

Verification of this statement boils down to finding such a sequence of rules. However, in order to correctly keep track of the conditions, Schmidt (2012) and Meier (2013) introduced a label for each rule which produces what they call *action facts* as a side-effect of the rule's application. This allows to encode "derive the same keys" and "correctly assume that each is talking to the other" by action facts. Such sequences of action facts are called *traces* when they additionally satisfy that no feasibility constraints are broken. For instance, each rule must be provided with the premise facts it consumes by the chain of previous rules.

Security properties such as the honest setup described above can then be implemented by first-order logic formulas on all traces, e.g. there exists a trace such that conditions on the action facts corresponding to the respective security property hold true.

This search for a satisfying trace is done by a backwards reachability analysis and constraint solving. Again simply put although inaccurately one can say that Tamarin-Prover does the following: the target action facts of an existentially quantified formula are the starting point. Then Tamarin-Prover analyses all possible paths the target action facts might arise from rule applications while satisfying the necessary conditions. This may give rise to other action or premise facts which are not yet resolved. The corresponding constraints are resolved–in the same way–by going further backwards until none is left. If this process terminates, Tamarin-Prover found a satisfying trace which means the statement is verified or is left with a contradictory constraint which means that there is no such trace and the statement is falsified.

Lemmata with universally quantified statements are verified or falsified by analysing the negated statement which is again an existentially quantified statement.

But there is no guarantee that Tamarin-Prover terminates during this process since the underlying decision problem is undecidable (Meier, 2013, sec. 9.4, pg. 152).

Unfortunately, a comprehensive discussion of the theoretical foundation of Tamarin-Prover go beyond the boundaries of this report. Please, confer to Meier (2013, ch. 8) for an in-depth look at the verification theory Tamarin-Prover implements.

Nevertheless, we will go into a little more detail about what facts and terms actually are because they are used very often.

Terms are the atomic entities in Tamarin-Prover. They represent cryptographic messages and bit strings, e.g. for key material. Mathematically, terms belong to a *order-sorted term algebra* which intuitively means that they come with a certain structure: They can be variables or constants of different *sorts* whereby the sorts have a partial order attached to them. In Tamarin-Prover, several sorts of terms are distinguished: message, public, fresh, and temporal terms. This is reflected in the syntax by a prefix in front of the variable and constant identifier. Message terms, the most common type of term, are not prefixed. Public terms are prefixed by `$`, fresh terms by `~`, temporal terms by `#`. For each sort, constant terms are enclosed in single quotes.

One can define *function symbols* over the sorts, meaning that the respective sort can be plugged in. Each function symbol has an arity. For instance, cryptographic primitives like encryption and decryption algorithms can be defined as binary/2-ary function symbols. Plugging the expected number of terms of appropriate sorts into a function symbol results in a term again. Suppose, there is a function symbol `enc` representing the encryption algorithm, then `enc(key, message)` is again a term. In Tamarin-Prover, we can define function symbols in the following way:

```
functions: enc/2, dec/2
```

Semantics of the function symbols can be described by *equational theories*. A general equation can be defined as follows:

```
equations: dec(key, enc(key, message)) = message
```

This equation–when grounded over all possible valuations of the two occurring terms `key` and `message`–induces an equational theory.

Now, *facts* are function symbols over the order-sorted term algebra described above under consideration of the equational theories given in the security protocol theory. They can be used to denote states and messages sent over a channel. There are three distinguished facts in Tamarin-Prover which come with certain constraints:

- The fresh fact `Fr` is used to represent random choice of its entry. That means, `Fr(~x)` establishes the fresh variable `~x` to be randomly chosen. The fresh fact is only allowed in the list of premise facts and can occur at most once for every variable.
- The fact `Out` represents messages sent over a public channel known to the attacker. Its content will be revealed to the attacker. However, the conclusion fact `Out(enc(k,x))` will only reveal `enc(k,x)` to the attacker. Neither `k` nor `x` will be given to the attacker which makes sense. `Out` can only occur in the list of conclusion facts of a rule.
- The fact `In` represents messages received over a public channel. They can only occur in the list of premise facts of a rule.

There are *message deduction rules* pre-defined in Tamarin-Prover. Among other things, they handle the transformation of `Out` facts into `In` facts so that these facts indeed represent sending and receiving messages. The message deduction rules include rules representing the manipulation abilities of the attacker who will gain knowledge over everything sent over the public channel this way.

This leaves us with the rough correspondences shown in table 1.

Key Exchange Protocol Implementation

The protocol we implemented is very brief: Alice, the client, chooses an ephemeral Diffie-Hellman private key a and composes a message: a quadruple consisting of the message type `client_hello`, an identifier of Alice, an identifier of the intended receiver and her public Diffie-Hellman key share $A = g^a$ where g is a pre-defined group element. Once Bob, the server, receives Alice's hello message, he himself chooses an ephemeral Diffie-Hellman secret key b, derives the shared secret $S = A^b = g^{ab}$, and sends a message consisting of the message type `server_hello`, Alice's identifier, his own identifier and his public Diffie-Hellman key $B = g^b$ to Alice. At this point, Bob

Language Constructs	Cryptographic Notions
Terms	Cryptographic messages, bit strings
Function Symbols	Cryptographic primitives
Equational Theories	Semantics of cryptographic primitives
Rules	State transitions of protocol instances, Oracles
Action facts	Protocol transcripts
Rewriting Systems	Protocol Specification, means of the Attacker
Traces	Parallel executions of the protocol
Trace Formulas	Security properties (e.g. confidentiality, authenticity)

Table 1: Correspondences between cryptographic notions and Tamarin-Prover's language

also creates a server session with the respective shared secret S. Once Alice receives Bob's message, she derives the shared secret $S' = B^a = g^{ab}$ herself and creates a client session with the derived shared secret S'.

We now want to describe the corresponding rules which implement this protocol. The basic syntax for a rule in Tamarin-Prover is as follows:

```
rule  Rule Identifier :
      [  Premise Facts  ]
      --[  Action Facts  ]->
      [  Conclusion Facts  ]
```

The rule identifier is used to refer to the rule, e.g. in error messages or in the interface of the interactive mode. The actual labeled rewriting rule is specified with an arrow notation, `[ ] --[ ]-> [ ]` where the first bracket specifies the premise facts which the rule consumes, the second optional bracket specifies the action facts which are created upon application of the rule and the third and last bracket is used for specifying the conclusion facts which are created by application of the rule.

Note that it is possible to create so-called persistent facts which are not consumed even though they are among the premise facts by prefixing an exclamation mark to the fact. This is used for creating identities, for instance. Some rules require that a certain identity, e.g. a server, exists while it is not desired that the existence of that identity disappears after some rule was applied. Actually this could also be realized by recreating the fact in the conclusion facts of all rules which also consume the fact. However, it would make the specification language unnecessarily complicated and prone to mistakes.

The first rule of the Diffie-Hellman key exchange we implemented is therefore a rule which creates the persistent identities:

```
rule create_identity:
    [] --> [ !Id($C) ]
```

This rule does not have any premise facts and establishes the existence of the persistent identity fact `!Id($C)` corresponding to a publicly known identifier `$C`, e.g. the IP address of the server or client.

Now, for the client. The client starts in an empty state. Once she decides to contact the server she compiles her hello message and transitions into a waiting state. This state has to hold the involved identities and the private key of the client. Therefore, we get the following rule:

```
rule client_hello:
    let
        A = 'g'^~a
    in
    [ !Id($C), !Id($S), Fr(~a) ]
    -->
    [ ClientWaiting($C, $S, ~a),
      Out(<'client_hello', $C, $S, A>) ]
```

This rule assumes that there are two known identities `$C` and `$S`. Moreover, using the fresh fact on the variable `~a` the client chooses a random private Diffie-Hellman key. In the conclusion of the rule we find the client waiting state–containing the session identities and the private Diffie-Hellman key–as well as the outgoing message, a quadruple of the message type `'client_hello'`, the identities again and the public Diffie-Hellman key `A` within the `Out` fact.

The server also starts in an empty state in which he expects a client hello message. Whenever such message arrives, he checks that the message type is correct and the identifier actually belongs to an existing identity. This is done by matching the identifier `C` to the persistent identity fact `!Id(C)`. More precisely, if `C` does not belong to a proper identity then the corresponding identity fact is not present for the premise facts of the rule. Then, the server chooses a private Diffie-Hellman key randomly by making use of the fresh fact on the variable `~b`, derives the corresponding public key `B = 'g'^~b` as well as the shared secret `secret = A^~b` using the data received from the client and creates a session with the identifiers and the shared secret. Finally, he composes a `server_hello` message and sends it using the `Out` fact again. This leaves us with the following rule:

```
rule server_receive_hello:
    let
        B = 'g'^~b
        secret = A^~b
    in
    [ !Id($S), !Id(C),
      In(<'client_hello', C, $S, A>),
      Fr(~b) ]
    --[ ServerCreatedSession(C, $S, secret) ]->
    [ ServerSession(C, $S, secret),
      Out(<'server_hello', C, $S, B>) ]
```

Note that the `Out` fact emitted by the `client_hello` rule is not explicitly converted into an `In` fact. This is because Tamarin-Prover automatically manages the means of the attacker. Since it implements a Dolev-Yao attack model, the adversary controls the traffic of the public channels. This is enforced by the message deduction rules which amongst others contain rules which convert `Out` facts into `In` and give the contents of the messages into the knowledge of the attacker, as explained earlier. Also, note that the action fact logs the creation of the server's session. This will later be used in the lemmata to analyse the security of this protocol.

The last rule implements the client receiving the `server_hello` message: It requires a `ClientWaiting` state in the premises, matching the identifiers and corresponding identities of the incoming message. The client derives the shared secret as well and creates a `ClientSession` with the same contents as the server did before. At this point the protocol is done. The rule looks as follows:

```
rule client_receive_hello:
    let
        secret = B^~a
    in
    [ !Id($C), !Id(S),
      ClientWaiting($C, S, ~a),
      In(<'server_hello', $C, S, B>) ]
    --[ ClientCreatedSession($C, S, secret) ]->
    [ ClientSession($C, S, secret) ]
```

Lemmata and Analysis

As said before, we analysed two lemmata for this version of a Diffie-Hellman key exchange. The first lemma ensures that the protocol as implemented is executable by honest parties and yields the expected result, i.e. both parties derive the same secret. This lemma can be written in Tamarin-Prover's syntax as follows:

```
lemma can_be_run:
    exists-trace
    "
        (Ex C S secret #i #j .
            ( ServerCreatedSession(C, S, secret) @ #i ) &
            ( ClientCreatedSession(C, S, secret) @ #j ) )
    "
```

Here, `exists-trace` indicates that the lemma named `can_be_run` is existentially quantified over the traces. Temporal variables are prefixed by a `#`. The lemma can be read as follows: There exists a trace–a sequence of rule applications–such that we can have a client `C`, a server `S` and a secret `secret` such that client and server created a session with those identifiers at moments `#i` and `#j` and both derived the same secret for this session.

The second lemma is intended to capture the confidentiality of the shared secret of the protocol. Simply put, the lemma states that when two parties establish a session with each other, the attacker does not obtain knowledge of the two secrets of the parties indicated by the `K` action fact. Here we restrict the traces to the interesting cases, namely that the server created his session before the client did and that server and client are different identities in the first place. The lemma in Tamarin-Prover's syntax reads as follows:

```
lemma man_in_the_middle:
    all-traces
    "
        All C S secret1 secret2 #i #j .
        (
            ServerCreatedSession(C, S, secret2) @ #j &
            ClientCreatedSession(C, S, secret1) @ #i &
            #j < #i &
            not(C = S)
        )
        ==>
        ( not(Ex #k1 #k2 .
            K(secret1) @ #k1 &
            K(secret2) @ #k2) )
    "
```

Note that the statement says that there are no moments `#k1` and `#k2` such that the attacker gained knowledge of `secret1` *and* `secret2`. This is a weaker security property than to say `secret1` *or* `secret2`. However, we will see that even the weaker security notion will not be verified by Tamarin-Prover. Also note that in an honest setup, we would have `secret1 = secret2`. But we do not want to restrict the powers of the attacker here: He might manipulate the messages causing the involved parties to derive different shared secrets. As we will see later, this is exactly what Tamarin-Prover discovers.

The complete source code of the introductory example is given in appendix A. When stored in a file named `DHKE.spthy`, one can analyse the protocol using Tamarin-

Prover using the command `tamarin-prover --prove DHKE.spthy`. This gives the following shortened result:

```
analyzed: DHKE.spthy

can_be_run (exists-trace): verified (11 steps)
man_in_the_middle (all-traces): falsified - found trace (11
  ↪  steps)
```

This tells us that a satisfying trace for the first lemma, `can_be_run`, could be found. However, the second lemma was falsified. That means that Tamarin-Prover found a counterexample to the statement. In other words, there is a way of applying the rules so that the attacker obtains knowledge of the respective secrets. This is a security leak. Figure 1 shows the constraint system of the attack against the second lemma which is provided when Tamarin-Prover is run in interactive mode. The green boxes show the

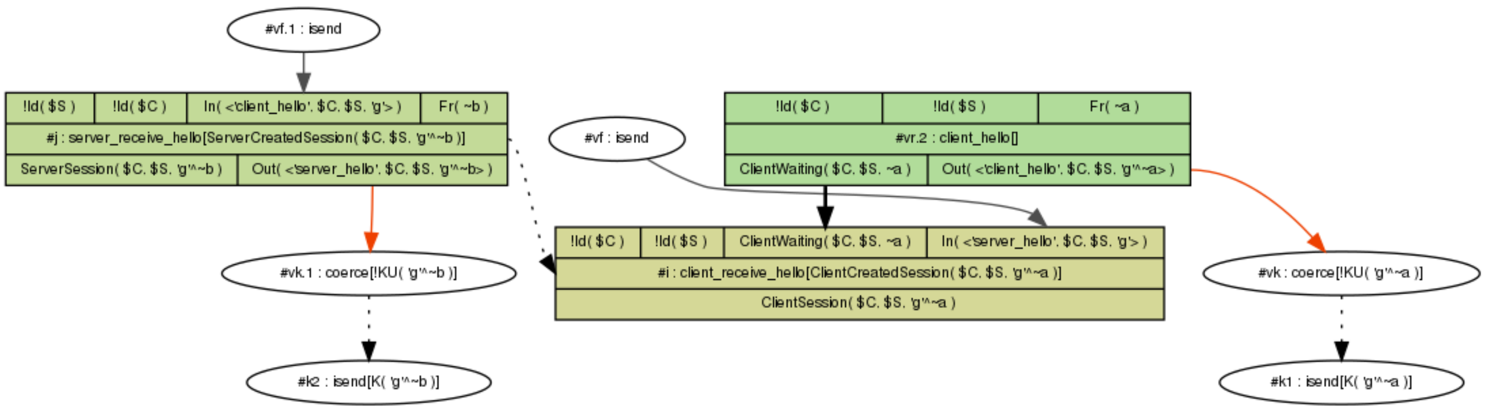
#vf.1 : isend
!Id($S)
!Id($C)
In(<'client_hello'. $C. $S. 'g'>)
Fr(~b)
#j : server_receive_hello[ServerCreatedSession($C. $S. 'g'^~b)]
ServerSession($C. $S. 'g'^~b)
Out(<'server_hello'. $C. $S. 'g'^~b>)
#vf : isend
!Id($C)
!Id($S)
Fr(~a)
#vr.2 : client_hello[]
ClientWaiting($C. $S. ~a)
Out(<'client_hello'. $C. $S. 'g'^~a>)
#vk.1 : coerce[!KU('g'^~b)]
!Id($C)
!Id($S)
ClientWaiting($C. $S. ~a)
In(<'server_hello'. $C. $S. 'g'>)
#i : client_receive_hello[ClientCreatedSession($C. $S. 'g'^~a)]
ClientSession($C. $S. 'g'^~a)
#vk : coerce[!KU('g'^~a)]
#k2 : isend[K('g'^~b)]
#k1 : isend[K('g'^~a)]

Figure 1: Constraint system after Tamarin-Prover found the counterexample for lemma `man_in_the_middle`

applied rules: the first line holds the premise facts, the second line the name of the rule and possibly the actions facts and the third line the conclusion facts. The actions of the attacker are shown as white ellipses. We can see that the attacker does not even let the server and the client actually communicate. He simply impersonates the other party, respectively, since identities as well as the used group element are not secured in any way and publicly known. Here, Tamarin-Prover confirms the well-known fact that Diffie-Hellman key exchanges without authentication do not provide any security in the face of an active attacker.

However, the Diffie-Hellman protocol as described and implemented in the previous section is most likely prone to DDoS attacks: The server creates and stores a session state when the handshake is not completed yet. Therefore, a client can open many different sessions with little effort and no need to store anything himself. This leverages the power of a malicious client to perform a DDoS attack on a server. However, Tamarin-Prover does not detect such vulnerabilities. At least, not without explicitly formulating a corresponding lemma which possibly makes it necessary to change the implementation, e.g. by introducing a session counter. This is a weakness of Tamarin-Prover's symbolic model.

2.2. Building Blocks for Cryptographic Protocols

Our main goal was to evaluate the suitability of Tamarin-Prover to analyse complex cryptographic protocols. In particular, we were interested in analysing the security of IPsec. We established the syntax and basic concepts of Tamarin-Prover in the previous section 2.1. When working towards a reference implementation of IPsec in Tamarin-Prover a crucial step is to implement and discuss the building blocks on which IPsec relies. The Internet Key Exchange v2 (IKEv2) used in IPsec in particular relies on random choices, Diffie-Hellman exponentiation, pseudo-random functions, signature schemes, authenticated encryption schemes, and certificates (Kaufman et al., 2014). Thus, we should discuss how it is possible to implement those in Tamarin-Prover and whether or not the chosen implementation reflects the intended notions.

Random Choices and Diffie-Hellman Exponentiation

Random choices and Diffie-Hellman exponentiation are built into Tamarin-Prover.

Random choices are made by using the `Fr` fact, the fresh fact. This is a special fact which is only allowed to appear at most once for each variable and only in the premise facts (Meier, 2013, pg. 77). This is enforced and checked by Tamarin-Prover in the well-formedness checks ran before the actual analysis. However, due to the way the symbolic model works it is very difficult to express the possibility and probability of two randomly chosen elements to be equal. A priori, two freshly generated names are different: "This fact [the `Fr` fact] must be used when generating fresh (random) values, and can only occur on the left-hand side of a rewrite rule, where its argument is the fresh term. Tamarin's underlying execution model has a built-in rule for generating instances of `Fr(x)` facts, and also ensures that each instance produces a term (instantiating `x`) that is different from all others." (The Tamarin Team, 2019, pg. 42).

It is reasonable to assert this property of the `Fr` fact for the sake of the symbolic model. Furthermore, collisions of randomly chosen elements in most cases have negligible probability and do not give an adversary any non-negligible advantage. However, one should keep in mind that this makes it impossible for Tamarin-Prover to detect insecure pseudo-random generators, for instance, because said property assumes the used pseudo-random generator to be absolutely secure. Again, most likely this is not a problem because Tamarin-Prover's security model treats cryptographic primitives as secure black boxes anyway.

Diffie-Hellman exponentiation was one of the contributions of Meier (2013). One can implement exponentiation rules by introducing an `exp` function symbol of arity 2. Expecting a base in the first entry and an exponent in the second. For a proper semantic, the exponentiation laws must also be given, e.g.

```
exp(exp(a,b),c) = exp(a,mult(b,c))
```

where `mult` represents multiplication, etc. Since Diffie-Hellman exponentiation is implemented in Tamarin-Prover exactly this way but also giving the exponentiation operator ^ as syntactic sugar it is advisable to use it as is.

Pseudo-Random Functions and Key Derivation Functions

Pseudo-random functions are used frequently in IPsec. For instance, the computation of an quantity called `SKEYSEED` relies on a negotiated pseudo-random function. From `SKEYSEED` all keys for the authentication phase of the key exchange are derived (Kaufman et al., 2014, sec. 1.2). Considering this we see that pseudo-random functions play a major role and their security is crucial.

In Tamarin-Prover there is only one way to represent cryptographic primitives and thus also pseudo-random functions. We have to define a function symbol for every such functions. For keyed pseudo-random functions, we would define them with arity two–one entry for the key and one entry for the input data:

```
functions: prf/2
```

The previous code defines a keyed pseudo-random function.

When defined this way, pseudo-random functions are idealized in the sense that a priori they do not have collisions. This is because in the symbolic model two different terms applied to the same function symbol yield two different terms again unless some explicitly defined equation demands otherwise. This makes Tamarin-Prover blind to some attacks which produce collisions and to attacks relying on a collision attack against the pseudo-random function.

Therefore, only pseudo-random functions which are considered to be secure should be analysed in Tamarin-Prover. Otherwise, the result Tamarin-Prover returns cannot be trusted.

The same holds for key derivation functions which are also defined using function symbols. For instance, in IPsec the quantity `SKEYSEED` is used to derive several keys for the initiator and the responder. In Tamarin-Prover, we can implement this by defining function symbols. Either one can define a key derivation function symbol with two entries–then the first entry might be used to determine which key is derived–or one defines multiple function symbols, one for each quantity derived from `SKEYSEED`. For the sake of code readability, we decided to define multiple function symbols.

Finally, there is another idealization which probably should be taken into account when it comes to function symbols, be it for pseudo-random functions or others. As said before, Tamarin-Prover handles cryptographic messages and inputs not as bit strings but as terms. Since terms are just symbols they do not have a length associated with them. This makes it more complicated to implement cipher suite negotiation. This is because in this case there might be two different protocol instances having negotiated different cipher suites. When the attacker decides in this situation to redirect messages from one protocol instance to another one, this might be detected in practice because the length requirements of the pseudo-random functions are not met in the other protocol instance. However, Tamarin-Prover does not have a notion of message length and would not detect this. This leaves two options: First, one accepts that in this case Tamarin-Prover does not accurately reflect the real scenario. At least, this makes the attacker stronger because he gets away with more than he should making this option conceivable. Second, one makes an effort to adjust the implementation so that Tamarin-Prover takes the message length into account. The latter proved to not be an easy task which is why we decided to take the first route.

Signature Schemes

A signature scheme Π consists of three algorithms: a key generator KeyGen, a signing algorithm Sign which signs messages and a verification algorithm Vrfy which checks a given signature and verifies or falsifies it for the signed document. For correctness, we require that

$$\mathsf{Vrfy}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = \mathsf{true} \tag{1}$$

for every message m and every secret and public key pair $(\mathsf{sk}, \mathsf{pk})$. In other words, a signature for message m created using the secret key sk should verify when checked against the corresponding public key pk.

As before, we implement the cryptographic primitives as function symbols. Therefore, we write

```
functions: sign/2, verify/3, pk/1, true/0
```

to define the signing and the verification algorithms. `pk` is a function symbol mapping a secret key to the corresponding public key and `true` is a 0-ary function symbol, i.e., a constant, representing the return value of `verify` when a signature is verified to be correct. Equation 1 can now be translated into Tamarin-Prover's language as follows:

```
equations: verify(pk(sk),m,sign(sk,m)) = true
```

Now, we have to deal with signature checking. Tamarin-Prover does not support if-statements controlling the conclusion facts depending on some condition on the premise facts, for instance. However, this is needed for signature checking because a party of a protocol instance might stop communicating when a signature does not verify but otherwise compile an appropriate answer message. We found that this can be solved in Tamarin-Prover in two possible ways:

In the first approach, one might write two rules having the same premise facts except for differing in the expected signature checking value. In particular, both rules expect the same protocol party's state. Now, one rule represents the case when the signature is evaluated to be invalid and leads into a failure state from which a protocol instance cannot be recovered. The second rule represents the case when the signature is evaluated to be valid and proceeds with the protocol. However, we found this approach overly complicated because it requires an additional axiom introducing a `false` value and it requires to be very careful about how this is handled. Therefore, we decided to use the following method which we believe to be equivalent in its effect.

In the second approach, one makes use of a language feature called *restrictions*. Using restrictions one can instruct Tamarin-Prover to disregard certain traces depending on the action facts. Let's consider, we write the following restriction:

```
restriction Equality:
    "All x y #i. Eq(x, y) @#i ==> x = y"
```

Then at every moment `#i` and for every entities `x` and `y` when the action fact `Eq(x,y)` arises, we require that `x` and `y` coincide. Otherwise, the whole trace is disregarded.

A rule using this method for signature checking might look as follows:

```
rule B_recv:
    [ !Pk(A, pk),
      In(<A, n, signature>) ]
    --[ Eq(verify(pk, n, signature), true),
        Verified(A, n) ]->
    [ ]
```

This rule cannot be applied when the signature of `n` does not verify against the public key `pk` because then the action fact `Eq(verify(pk, n, signature), true)` does not satisfy the restriction and a corresponding trace would be disregarded as a whole.

This approach has the advantage to shift the signature checking from premise and conclusion facts to action facts which is an area the attacker does not have any control over. This is desired because a malicious attacker might manipulate the signature of the message or the message itself but as soon as both arrived at the recipient in the protocol instance the attacker is not able to manipulate the signature checking algorithm. Hence, this reflects reality more accurately than the earlier approach.

We implemented an exemplary protocol using signature checking: Alice sends a signed nonce to Bob who checks the signature. The code for this toy example can be found in appendix B.

Authenticated Encryption Schemes and Certificates

The implementation of the authentication phase of IKEv2 in IPsec went beyond the scope of our lab. Nevertheless, we want to discuss authenticated encryption schemes and certificates which are relevant for the authentication phase of IKEv2 very briefly. What follows, however, should not be seen as an experiential report but as a theoretical remark since we did not implement the ideas.

As explained earlier, the `Out` and `In` facts are for message sending and receiving. Everything sent using these facts can be manipulated by the attacker who also obtains knowledge about everything in those facts. Additionally, Tamarin-Prover also provides an authenticated channel with corresponding `Out_A` and `In_A` facts. While the attacker also obtains knowledge about everything sent over this channel, he does not have the ability to manipulate the messages. This again is handled by the message deduction rules. Sending a ciphertext via this channel might be seen as an authenticated encryption. However, this does not reflect the real scenario accurately. Therefore, we would advice against choosing this route.

However, from the earlier discussion one might already get the impression how an authenticated encryption might be implemented. An encrypt-then-authenticate (EtA) mechanism could easily be implemented by using function symbols and restrictions as demonstrated before. This can be used for an initial step towards implementing authenticated encryption. Nonetheless, for the sake of cipher suite negotiation, i.e. independence of the choice of cryptographic primitive, it is necessary to generalize this mechanism.

When it comes to certificates they are basically a collection of identities, public keys, signatures, and possibly some additional information and references like the applied algorithms. One of the most common formats of certificates is the X.509 standard (Cooper et al., 2008). Since this is mainly a concern of data organization it is very likely that this format could be implemented in Tamarin-Prover or at least to a very large degree. It should also be possible to implement a public key infrastructure with certification authorities issuing certificates using labeled rewriting rules and restrictions ensuring the validation of certificates.

An implementation of the X.509 standard is of high utility for Tamarin-Prover in general because of the wide spreading of it. A well-designed implementation of the X.509 standard could be reused in the analyses of lots of protocols using it.

Cremers et al. (2017) used the public key as a certificate in rev21 of their code. The matching of identity to the public key was done via the structure of the messages

(Cremers et al., 2018).

3. Towards an Automatic Analysis of IPsec

Before we dive into the implementation and analysis of the initialization phase of IKEv2 in IPsec we briefly describe the key exchange protocol (Kaufman et al., 2014). Simply put, one can divide IKEv2 into two phases: an initialization phase and an authentication phase. The purpose of the initialization phase is to execute a Diffie-Hellman key exchange between initiator/client and responder/server in order to derive a quantity called `SKEYSEED`. This in turn is used to derive keys to secure the authentication phase with authenticated encryption. The authentication phase–as the name suggests–is done to authenticate client and/or server using their certificates and to derive key material `KEYMAT` which is then used to encrypt the further communication between the involved parties.

When the initiator decides to establish a connection to the responder, she starts the initialization phase by sending the first message $m_1 = \langle \texttt{Hdr}, \texttt{SAi1}, \texttt{KEi}, \texttt{Ni} \rangle$:

- The header `Hdr` contains the "security parameter indices (SPIs), version numbers, Exchange type, Message ID, and flags of various sorts" (Kaufman et al., 2014). Simply put, the security parameter index identifies a connection.
- The first security association of the initiator `SAi1` lists the supported algorithms for encryption/decryption, authentication, pseudo-random functions, groups for the Diffie-Hellman key exchange, and a pseudo-random function.
- The key exchange material of the initiator `KEi` contains a randomly chosen exponent after the initiator guessed the cipher suite negotiation for the responder.
- A randomly chosen nonce `Ni` to be used in the key derivation and authentication.

When the responder receives the message m_1 he first checks whether the guessed cipher suite negotiation is supported or not. Afterwards, he compiles a message $m_2 = \langle \texttt{Hdr}, \texttt{SAr1}, \texttt{KEr}, \texttt{Nr}, \texttt{CERTREQ} \rangle$:

- The header `Hdr` is built as described before.
- `SAr1` now contains the chosen algorithms which concludes the cipher suite negotiation.
- `KEr` contains the key exchange material of the responder, e.g. a randomly chosen exponent in the chosen group.
- `Nr` is the nonce of the responder.

After m_1 and m_2 are exchanged, both derive $\texttt{SKEYSEED} = \text{prf}(\texttt{Ni}|\texttt{Nr}, \texttt{DH})$ where `DH` is the shared secret from the Diffie-Hellman key exchange and prf denotes the chosen pseudo-random function. This concludes the initialization phase.

Since we did not implement the authentication phase, we omit a detailed description for the sake of simplicity. More details on this part of the protocol can be obtained from

Kaufman et al. (2014).

3.1. Finite State Machine of IKEv2 Initialization Phase

For our implementation of the IKEv2 initialization phase, we decided on the finite state machine depicted in figure 2. Our code is available online via GitHub (Stadtländer, 2019).

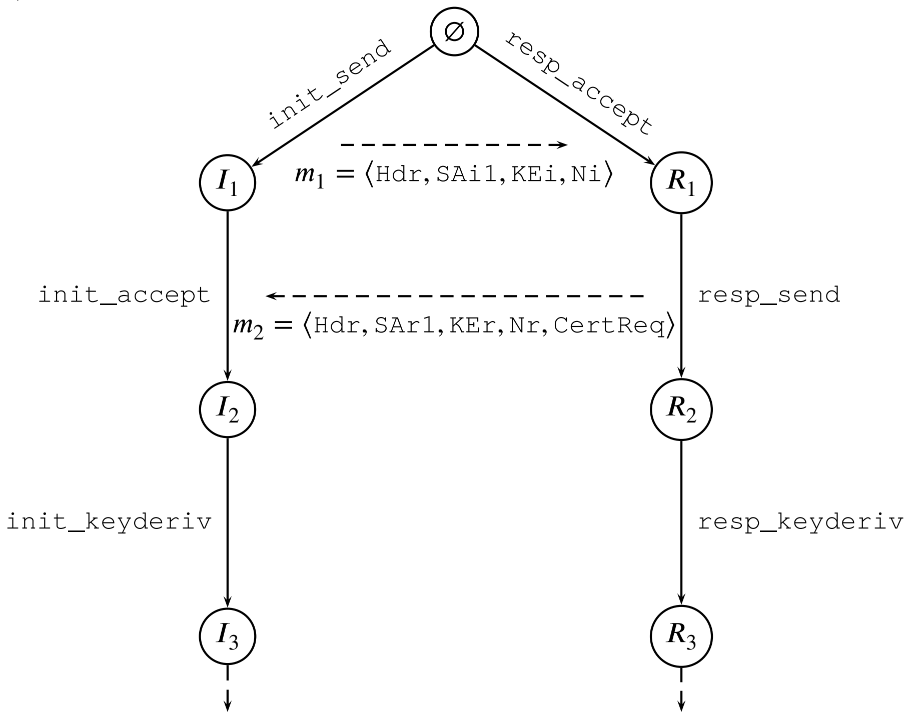

Figure 2: Finite state machine of the initialization phase of IKEv2 (failure states omitted)

Both the responder and the initiator start off in an empty state. The state as we implemented it is initialized with 0 and contains fields for all quantities relevant for the key exchange: roles, identifiers, security parameter indices, Diffie-Hellman parameters, nonces and derived keys. Once the initiator decides to send the first message m_1, the rule `IKE_SA_INIT_initiator_send` is applied. Note that all rule names are abbreviated in figure 2 for a better visualization. As conclusion facts of the rule, the state `State_I1` of the initiator and an `Out` fact with the outgoing message are created.

The rule `IKE_SA_INIT_responder_accept_g1` is meant to accept the message coming from the initiator choosing group element `g1`. This also produces a state `State_R1` for the responder. However, when the initiator did not choose `g1` as a group element, the only option is to apply the rule `IKE_SA_INIT_responder_reject` which creates a failure state `State_RX`.

When the state `State_R1` is present, the rule `IKE_SA_INIT_responder_send` can be applied to compose and send message m_2 and to create the following responder state `State_R2`.

The initiator receives message m_2 when rule `IKE_SA_INIT_initiator_accept` is applied. She transitions into the following state `State_I2`. In states `State_I2` and `State_R2` both can use the data in the states to derive `SKEYSEED`.

Note that along the state transitions we described above, all kinds of sanity checks are performed to ensure that everything is as negotiated, for instance.

3.2. Implementation Details

We want to give a brief overview over the structure of the code and some additional details to our implementation of the initialization phase of IKEv2.

Since Tamarin-Prover by nature does not support security protocol theories to be spread over several files, i.e. there is no such thing as an include command, we made use of the m4 macro preprocessor inspired by Cremers et al. (2017). The root file is `ipsec.m4`. Then we have the following files:

- `state.m4i` defining the macros and settings connected to the states of the initiator and the responder.
- `restrictions.m4i` contains all restrictions used in our code. In particular, the equality restriction we already described in the context of signature schemes in section 2.2.
- `pki.m4i` describes the the public key infrastructure. In our case, this is the identity creation.
- `attacker.m4i` contains the means of the attacker: currently, this is a corrupt oracle which can be used to reveal the long-term secret of the identities.
- `model_initiator.m4i` and `model_responder.m4i` contain the rules for the initiator and the responder, respectively.
- Finally, `lemmas.m4i` states the lemmata about the key exchange to be analysed by Tamarin-Prover.

The states of the initiator and the responder are of the same format. They both contain the role (initiator or responder), identifiers of the respective parties, security parameter indices, Diffie-Hellman parameters/keys/secrets, nonces and the derived keys. Therefore, we used the same macros for creating, reading, updating and writing states for both: A `init_state` macro was used to create a null-initialized state, a `set_state` macro was used to parse the previous state from the premise facts to the current variables, and a `next_state` macro was used to write the new state to the conclusion facts. The skeleton of a rule for the initiator thus looked as follows:

```
rule IKE_SA_INIT_initiator_*:
    let
        set_state()
        /* ... */
    in
    [ State_Ix(prev_state())
      /* ... */ ]
    -->
    [ State_Iy(next_state())
      /* ... */ ]
```

This procedure enabled us to only describe the differences of the state in the transition instead of listing every field in the state explicitly. This makes the code more robust against mistakes and improves the readability thereof.

We implemented a very rudimentary version of the IPsec key exchange, so far. In particular, we do not support real cipher suite negotiation: While the initiator is programmed to offer two group elements in the Diffie-Hellman key exchange, the responder only supports one of them, called `g1`.

3.3. Lemmata and Results

We analysed two lemmata for the initialization phase of IKEv2. The first being called `honest_init`. Similar to what we did for the Diffie-Hellman key exchange in section 2.1, this lemma checks whether or not it is possible to execute the protocol with honest parties so that both parties derive the same keys. This can be seen as a sanity check for the implementation and indeed was verified by Tamarin-Prover.

The other lemma we stated deals with the question whether or not the security parameter index as a sole identification for the connection is secure: Can an attacker create a situation where two parties have the same pair of security parameter indices but derive different secrete keys? We stated the lemma as follows:

```
lemma security_parameter_not_secure:
    exists-trace
    "
        Ex SPIi SPIr SK1 SK2 #i #j .
        (
            ResponderKeyDerived(SPIi, SPIr, SK1) @ #i &
            InitiatorKeyDerived(SPIi, SPIr, SK2) @ #j &
            not(SK1 = SK2)
        )
    "
```

and Tamarin-Prover verified it as well. According to the constraint system obtained from Tamarin-Prover's interactive mode the attacker achieves this by doing the following: He impersonates the initiator by using the security parameter index sent by the initiator but choosing every other parameter in the message sent to the responder.

This motivates the authentication phase of IPsec's key exchange and in particular the signing of the early message m_1 and the received nonce. The changes made by the attacker would be detected there causing the party to abort the protocol.

4. Discussion

4.1. Tamarin-Prover's Security Model

We stated in the introduction that we wanted to evaluate Tamarin-Prover's suitability to analyse the security of large protocols suites such as IPsec. More generally, we wanted to evaluate the question "Can Tamarin-Prover improve the trustworthiness of cryptographic protocols".

According to our lab experiences, the answer to this question is a conditional yes. Tamarin-Prover does not give an absolute notion of security but more like an own category of Tamarin-security, say "This protocol is Tamarin-secure".

The reason for this lies in the Dolev-Yao attack model as well as the symbolic model in which Tamarin-Prover operates. The Dolev-Yao attack model implements an attacker who

1. controls the network traffic, i.e. may intercept, manipulate, and redirect messages, and
2. has access to a *reveal oracle*, i.e. can obtain the long-term secrets of all involved parties.

Via the message deduction rules and the way in which `In` and `Out` facts are intertwined with the message deduction rules Tamarin-Prover indeed implements a Dolev-Yao attacker. However, the reveal oracle has to be implemented by the user. While this gives the flexibility of tweaking the Dolev-Yao attack model as needed, it adds additional effort on the user-side which may be prone to mistakes or improper implementation.

Since Tamarin-Prover sees cryptographic messages not as binary strings but as terms over a term algebra and cryptographic primitives not as algorithms but as function symbols, it does not work in the well-known computational model similar to the perspective of EasyCrypt but it has its own model the authors refer to as the *symbolic model*. This determines the security which Tamarin-Prover analyses: It is blindfold with respect to more fine-grained attacks like collision attacks against pseudo-random functions or also against Bleichenbacher-type attacks as we have seen in section 2.2.

Moreover, Tamarin-Prover is also blindfold with respect to practical attacks like DDoS as argued in the context of the Diffie-Hellman key exchange protocol, section 2.1.

4.2. Practical Limitations of Tamarin-Prover

However, Tamarin-Prover also has some practical limitations. The particular syntax it uses to implement protocols and lemmata make it very difficult to translate the desired security properties from the established mathematical notions into the symbolic model.

The backwards reachability analysis underpinning Tamarin-Prover may a) create exponentially many paths with respect to the number of constraints to be solved and b) prevent Tamarin-Prover from terminating because the underlying statement verification problem is undecidable. Thus, memory and running time necessary to analyse cryptographic protocols with Tamarin-Prover may become a serious bottleneck.

4.3. Strengths of Tamarin-Prover

But this does not render Tamarin-Prover being useless. The symbolic model shines for what it was made for: interaction attacks. Interaction attacks exploit the ways in which the cryptographic primitives and concurrently executed protocol instances *interact*, hence the name. A well-known type of interaction attacks are replay attacks where a malicious adversary may impersonate another identity by replaying already sent messages to the server.

This explains why the symbolic model of Tamarin-Prover is such a good fit for detecting this type of attacks: First, the symbolic model treats all primitives as function symbols, i.e. black boxes which are agnostic about their implementation or the format of the messages. This simplification makes it possible for Tamarin-Prover to reason about how the rules, function symbols and terms behave when they are plugged together in unintended ways. Second, Tamarin-Prover supports unbounded parallel execution of arbitrarily many protocol instances (Meier, 2013). These two ingredients are tailored to interaction attacks.

5. Conclusion and Future Work

After having implemented the initialization phase of the key exchange protocol used in IPsec there are several routes open for future work. First, the negotiation of cipher suites was not implemented in our code. Second, after the initialization phase comes the authentication phase as a crucial part of the key exchange. Also, the corresponding security properties need to be analysed. Third, we did not implement certificates. It is potentially useful for many protocols if there was an implementation of a formatting standard of certificates such as X.509.

However, Tamarin-Prover does not provide language features for code re-usability, e.g. there is no include command. This is one practical aspect of improving Tamarin-Prover making it possible to easily extend implementations of other users, e.g. including the X.509 implementation mentioned before. Another aspect would be to introduce some kind of unit testing for rules to Tamarin-Prover's language to prevent implementation mistakes and reduce the gap between protocol specification and implementation even further.

In conclusion, we can say that Tamarin-Prover proved itself being very useful for analysing the security of protocols used in large-scale multi-party networks. It is not suited for analysing cryptographic primitives since they are treated as black boxes in the symbolic model. The symbolic model appears to be only appropriate to detect interaction attacks. Therefore, the results gained from Tamarin-Prover should not be taken as a sole measure of a protocol's security. The security notion it represents can be a very good complementary layer of trust additional to other means of analysis, e.g. by security proofs by hand or other automatic analyses. Particularly, due to the nature of the symbolic model, the used cryptographic primitives should be very carefully analysed beforehand because Tamarin-Prover implicitly assumes their security. Apart from this constraint, we found Tamarin-Prover suited and useful as an analysis tool even for large protocols like TLS or IPsec.

References

Basin, David, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse, and Benedikt Schmidt (Mar. 10, 2014). *Tamarin Prover*. URL: `https://tamarin-prover.github.io/` (visited on Jan. 18, 2019).

Bhargavan, Karthikeyan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub (2013). "Implementing TLS with verified cryptographic security". In: *IEEE Symposium on Security and Privacy*. IEEE, pp. 445–459.

Bhargavan, Karthikeyan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin (2014). "Proving the TLS Handshake Secure (as it is)". In: *Advances in Cryptology – CRYPTO 2014*. Ed. by Juan A. Garay and Rosario Gennaro. Springer Berlin Heidelberg, pp. 235–255. URL: `https://eprint.iacr.org/2014/182` (visited on May 13, 2018).

Bleichenbacher, Daniel (1998). "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1". In: *Advances in Cryptology — CRYPTO '98*. Ed. by Hugo Krawczyk. Springer Berlin Heidelberg, pp. 1–12.

Blum, Norbert (Aug. 11, 2017). "A Solution of the P versus NP Problem". In: *arXiv e-prints*. Withdrawn. arXiv: `1708.03486v2 [cs.CC]`. URL: `http://arxiv.org/abs/1708.03486` (visited on Jan. 18, 2019).

Cooper, D., S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk (2008). *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List*

(CRL) Profile. RFC 5280. RFC Editor. URL: `http://www.rfc-editor.org/rfc/rfc5280.txt` (visited on Feb. 2, 2019).

Cremers, Cas, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe (2017). “A Comprehensive Symbolic Analysis of TLS 1.3”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. ACM, pp. 1773–1788. URL: `http://doi.acm.org/10.1145/3133956.3134063`.

– (Sept. 27, 2018). *TLS13Tamarin*. URL: `https://github.com/tls13tamarin/TLS13Tamarin` (visited on Feb. 2, 2019).

Durumeric, Zakir, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman (2014). “The Matter of Heartbleed”. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC ’14. Vancouver, BC, Canada: ACM, pp. 475–488. URL: `http://doi.acm.org/10.1145/2663716.2663755` (visited on Jan. 17, 2019).

Gonthier, Georges (2008). “Formal proof–the four-color theorem”. In: *Notices of the AMS* 55.11, pp. 1382–1393.

IMDEA Software Institute (2009). *EasyCrypt*. URL: `https://www.easycrypt.info` (visited on Jan. 18, 2019).

Kaufman, C., P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen (2014). *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC 7296. RFC Editor. URL: `http://www.rfc-editor.org/rfc/rfc7296.txt` (visited on Feb. 1, 2019).

Lindell, Yehuda and Jonathan Katz (2014). *Introduction to Modern Cryptography*. 2nd ed. Chapman and Hall/CRC.

Meier, Simon (2013). “Advancing automated security protocol verification”. Dissertation. ETH Zürich.

MITRE, ed. (Dec. 3, 2013). *CVE-2014-0160*. CVE-ID. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160` (visited on Jan. 17, 2019).

Mochizuki, Sinichi (June 28, 2018). *Inter-universal Teichmüller Theory I-IV*. URL: `http://www.kurims.kyoto-u.ac.jp/~motizuki/papers-english.html` (visited on Jan. 18, 2019).

Nussbaumer, Jakob and Michael Nüsken (2018). “Cryptographic Game-style language in EasyCrypt”. In: *crypto day matters 29*. Ed. by Christopher Huth and Michael Nüsken. Bonn: Gesellschaft für Informatik e.V. / FG KRYPTO.

Opfer, Gerhard (2011). “An Analytic Approach to the Collatz 3n+1 Problem”. In: *Hamburger Beiträge zur Angewandten Mathematik* 9.

Schmidt, Benedikt (2012). “Formal analysis of key exchange protocols and physical protocols”. Dissertation. ETH Zürich.

Stadtländer, Eike (Feb. 2, 2019). *IPsecTamarin*. URL: `https://github.com/estadtlaender/IPsecTamarin` (visited on Feb. 2, 2019).

The F* Team (2019). *Verified Programming in F*. A tutorial*. English. MSR-Inria. URL: `https://www.fstar-lang.org/tutorial/` (visited on Jan. 24, 2019).

The Tamarin Team (Jan. 18, 2019). *Tamarin-Prover Manual. Security Protocol Analysis in the Symbolic Model*. URL: `https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf` (visited on Jan. 25, 2019).

Wong, David (June 14, 2017). *Tamarin Prover Introduction*. URL: `https://www.youtube.com/watch?v=XptJG19hDcQ` (visited on Jan. 24, 2019).

A. Code of the Introductory Example

The following is the full source code of the introductory example described in section 2.1. This might be stored in a file called `DHKE.spthy` and analysed with Tamarin-Prover by calling `tamarin-prover --prove DHKE.spthy`:

```
theory DHKE
begin

builtins: diffie-hellman

rule create_identity:
    [] --> [ !Id($C) ]

rule client_hello:
    let
        A = 'g'^~a
    in
    [ !Id($C), !Id($S), Fr(~a) ]
    -->
    [ ClientWaiting($C, $S, ~a),
      Out(<'client_hello', $C, $S, A>) ]

rule server_receive_hello:
    let
        B = 'g'^~b
        secret = A^~b
    in
    [ !Id($S), !Id(C),
      In(<'client_hello', C, $S, A>),
      Fr(~b) ]
    --[ ServerCreatedSession(C, $S, secret) ]->
    [ ServerSession(C, $S, secret),
      Out(<'server_hello', C, $S, B>) ]

rule client_receive_hello:
    let
        secret = B^~a
    in
    [ !Id($C), !Id(S),
      ClientWaiting($C, S, ~a),
      In(<'server_hello', $C, S, B>) ]
    --[ ClientCreatedSession($C, S, secret) ]->
    [ ClientSession($C, S, secret) ]

lemma can_be_run:
    exists-trace
    "
        (Ex C S secret #i #j .
            ( ServerCreatedSession(C, S, secret) @ #i ) &
            ( ClientCreatedSession(C, S, secret) @ #j ) )
    "

lemma man_in_the_middle:
    all-traces
    "
        All C S secret1 secret2 #i #j .
        (
            ServerCreatedSession(C, S, secret2) @ #j &
            ClientCreatedSession(C, S, secret1) @ #i &
            #j < #i &
            not(C = S)
        )
        ==>
        ( not(Ex #k1 #k2 .
            K(secret1) @ #k1 &
            K(secret2) @ #k2) )
    "

end
```

B. Code of the Signature Scheme Example

The following security protocol theory implements a protocol which uses signature checking as described in section 2.2. For the protocol, we assume to have a public key infrastructure so that every party has uncompromised access to a public key of every identity. This is realized in the `create_identity` rule. However, following the Dolev-Yao attack model, we provide a corrupt oracle to the attacker so that he can obtain the secret long-term key of any identity in the public key infrastructure by applying the rule `corrupt`.

The actual protocol works as follows: Alice chooses a nonce `n` randomly, signs it using her private key and sends a message containing her identity, the nonce and the signature to Bob. Upon receiving, Bob checks the signature of Alice and either verifies or falsifies the incoming message.

The lemma–which is verified by Tamarin-Prover–states that whenever Bob verifies a message from Alice, either the attacker used the corrupt oracle or Alice indeed sent the message.

```
theory SimpleAuth
begin

functions: sign/2, verify/3, pk/1, true/0

equations: verify(pk(sk), m, sign(sk, m)) = true

restriction Equality:
    "All x y #i. Eq(x, y) @#i ==> x = y"

rule create_identity:
    [ Fr(~ltkI) ]
    -->
    [ !Id($I, ~ltkI), !Pk($I, pk(~ltkI)),
      Out(pk(~ltkI)) ]

rule corrupt:
    [ !Id($I, ~ltkI) ]
    --[ Corrupted($I) ]->
    [ Out(~ltkI) ]

rule A_send:
    [ Fr(~n), !Id(A, ltkI) ]
    --[ Sent(A, ~n) ]->
    [ Out(<A, ~n, sign(ltkI, ~n)>) ]

rule B_recv:
    [ !Pk(A, pk),
      In(<A, n, signature>) ]
    --[ Eq(verify(pk, n, signature), true),
        Verified(A, n) ]->
    [ ]
```

```

lemma authenticity:
    all-traces
    "
        All A n #i.
        (
            Verified(A, n) @ #i
            ==>
            (
                (Ex #j . (Sent(A, n) @ #j & #j < #i)) |
                (Ex #k . (Corrupted(A) @ #k & #k < #i))
            )
        )
    "

end
```