DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, *SWEDEN 2015*

Verification of security protocols with state in ProVerif

AVOIDING FALSE ATTACKS WHEN VERIFYING FRESHNESS

PASI SAARINEN

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

Verification of security protocols with state in ProVerif

Avoiding false attacks when verifying freshness

Verifiering av säkerhetsprotokoll med persistenta variabler i ProVerif

Att undvika falska attacker vid verifiering av att genererade nycklar är unika

Pasi Saarinen <pasis@kth.se>

School of Computer Science and Communication
Royal Institute of Technology
Thesis topic: Computer Science
Program: Civilingenjör Datateknik
Supervisor at CSC: Dilian Gurov
Examiner: Mads Dam

Provider of the Project: Ericsson AB
Supervisors at Ericsson AB: Noamen Ben Henda, Karl Norrman

June 30, 2015

Abstract

One of the issues when attempting to verify security properties of a protocol is how to model the protocol. We introduce a method for verifying event freshness in tools which use the applied π-calculus and are able to verify secrecy. Event freshness can be used to prove that a protocol never generates the same key twice. In this work we encode state in the applied π-calculus and perform bounded verification of freshness for MiniDC by using the ProVerif tool. MiniDC is a trivial protocol that for each iteration of a loop generates a unique key and outputs it to a private channel. When verifying freshness, the abstractions of ProVerif cause false attacks. We describe methods which can be used to avoid false attacks that appear when verifying freshness. We show how to avoid some false attacks introduced by private channels, state and protocols that disclose their secret. We conclude that the method used to verify freshness in MiniDC is impractical to use in more complicated protocols with state.

Sammanfattning

Ett av problemen som uppstår vid verifiering av säkerhetsprotokoll är hur protokoll ska modelleras. Vi introducerar en metod för att verifiera att skapde termer inte har använts förr. Denna metod kan användas i program som använder applicerad π-kalkyl som input och kan verifiera sekretess. I detta arbete visar vi hur protokoll med persistenta variabler kan modelleras i applicerad π-kalkyl. Vi verifierar även MiniDC för ett begränsat antal iterationer med hjälp av ProVerif. MiniDC är ett enkelt protokoll som för varje iteration av en loop skapar en nyckel och skickar den över en privat kanal. När man verifierar att skapade termer inte har använts förr så introducerar ProVerifs abstraktioner falska attacker. Vi beskriver metoder som kan användas för att undvika dessa falska attacker. Dessa metoder kan användas för falska attacker introducerade av privata kanaler, persistenta variabler eller protokoll som avslöjar sin krypteringsnyckel. Vår slutsats är att metoden som används för att verifiera MiniDC är opraktisk att använda i mer komplicerade protokoll med persistenta variabler.

Contents

Chapter 1

Introduction

Computers are integrated into every aspect of society today. Almost all computers perform some sort of communication with another computer. As communication is done over insecure channels such as the internet or wireless transfer, there is a need for protocols that use insecure channels for secure communication. These protocols are called security protocols.

Security protocols often come with some kind of guarantee that an adversary cannot modify the outcome of communication. A common method to argue that a security protocol works is by using informal arguments. This method has been shown to be error prone. As an example, the Needham-Schroeder [1] protocol had an informal argument of correctness and was thought to be secure for almost two decades. In 1996 this was disproven when Lowe found an attack on the protocol [2]. A more recent example is an attack on Google's Single-Sign-On protocol which allowed a service provider to impersonate a user for another service provider [3].

Another method that can be used to show that a protocol works in the presence of an adversary is formal verification. Formal verification of a protocol results in a proof that it is impossible for an adversary to break certain properties of the protocol. Formal verification may also result in a proof that an attack exists. The attacks on the protocols above were found during formal verification.

To formally verify a protocol, a model of the adversary and the network has to be adopted. This model is simply a specification of what an adversary can do and how the network behaves. Many models have been employed for this task but most of them are variations of the Dolev-Yao [4] model. This model assumes that cryptographic primitives are unbreakable and that the adversary has full control over all communication.

There are many problems with formal verification of security protocols. For example there are several undecidability results [5, 6, 7]. These results show that it is an undecidable problem to verify that a key used in a protocol is kept secret. It is not only an undecidable problem for general protocols but also for protocols with limitations on for example message length. Another issue is the fact that manual proofs often are complex and long.

Tools for automatic verification can mitigate certain problems of manual formal verification. Such tools limit the problem of human error to the design of the tool and description of the protocol. Tools can also produce proofs in a format which allows the proof to be checked both automatically and manually. Because of the benefits of automatic verification, research on tools for automatic verification is of interest.

When formally verifying a security protocol, the applied π-calculus [8] can be used. This calculus allows one to specify a protocol and the encryption primitives it uses. Applied π-calculus is used as the input language for some verification tools. One problem that arises from the use of the applied π-calculus is how to model a protocol.

Protocols are often specified and implemented with imperative programming in mind, which can be hard to translate into the applied π-calculus. It is not always easy to describe protocols in the applied π-calculus, and it can be hard to verify that the representation corresponds to the specification of the protocol. This thesis studies how protocols can be translated into the applied π-calculus and verified with the ProVerif tool.

We focus on how to encode state in the applied π-calculus in a manner that causes ProVerif to terminate without false attacks. As state is introduced, a property called freshness becomes more important. Because of this we evaluate ways to model and prove freshness when using the applied π-calculus in ProVerif.

1.1 Motivation

Ericsson research works with protocols that are complex and have not been formally verified yet. Some protocols are complex because of the limitations of the hardware they are implemented on. Other protocols may not be limited by the hardware but are standardized. In both cases we cannot modify the protocol to use structures that are easy to verify.

We need to analyse how to model the different structures of such protocols in the current state-of-the-art tools. From Ericsson's point of view the two tools of interest were Tamarin [9] and ProVerif [10]. We deemed Tamarin to be in an early development stage as the documentation was sparse and non-termination seemed to be a common problem. This led us to focus on ProVerif and wait with attempts to use Tamarin until the tool is more mature. We later uncovered a bug in Tamarin that resulted in a property that did not hold being verified as true.[1]

One protocol of specific interest for Ericsson has been the Dual Connectivity (DC) protocol [11, Annex E]. DC seems to be hard to verify using current state-of-the-art tools. At Ericsson, one attempt at verifying this protocol has been made [12]. This attempt did not manage to prove key freshness for an unbounded number of sessions.

[1] https://github.com/tamarin-prover/tamarin-prover/issues/144

1.2 Goal

The goal of this thesis is to create constructs that could be helpful when verifying the DC protocol. We want the constructs to be general so that they can be applied when verifying other protocols with similar structures.

The two constructs we have identified to be most important are the notion of state and freshness. An example of a protocol with state is the protocol describing a server that assigns an increasing identity number to each client based on the order they connect. The last output identity can be considered to be the state of the server and must be remembered between different clients. If the identity that is output is unique for each client, then the identity is fresh and the protocol achieves freshness of the client identity.

The DC protocol uses counters to avoid keystream reuse. Keystream reuse is often an unwanted property as it may allow the adversary to perform attacks on the protocol. We want to be able to prove freshness for the generated keystream, and need to model state to describe the counter.

1.3 Method

The identification of encodings for freshness and state is done by studying previous work and the applied π-calculus. These encodings are then tested in ProVerif to identify how the abstractions of ProVerif interact with them. An iterative and experimental approach is employed when identifying and rectifying false attacks on freshness which are caused by the encodings. If encodings that can be used to verify freshness with state, these are applied to a case study. This case study is used to test performance and usability of the encodings. Conclusions are drawn from the results of the case study.

1.4 Delimitations

As we focus on ProVerif we are subject to its limitations and features. We only consider translations into a version of applied π-calculus which is supported by ProVerif. For example, recursion is not supported by ProVerif.

We do not consider the notion of observational equivalence and make no analysis if the encodings or ways to avoid false positives, break this property.

No proofs of correctness are created although we will try to give an intuitive argument for why the proposed encodings work as intended.

1.5 Structure

The thesis begins with an introduction that describes the problem and our approach to solve it. This is followed by a background chapter that presents information needed to understand the work that has been done. The background gives a detailed

description of the version of applied π-calculus used in this work. It also discusses ProVerif and its abstractions, formal verification and security protocols in general.

Chapter three defines encodings of state and event freshness in the applied π-calculus. During this chapter we give little consideration to how the encodings interact with the abstractions of ProVerif.

Chapter four explains how to avoid false attacks when verifying freshness. We focus on the false attacks caused by the encoding of state.

In chapter five we use the encodings and methods from the previous chapters to verify event freshness in a trivial protocol with state. This case study is used to test the performance and usability of the methods and encodings.

Finally, in chapter six we draw conclusions, discuss our results and propose future work.

Chapter 2

Background

In this chapter we present the background knowledge needed to understand the rest of the thesis. The thesis targets formal verification of security protocols and therefore we describe both security protocols and formal verification of such.

We explain the language we code protocols in, the applied π-calculus. The tool that we use for verification is ProVerif and a detailed description of it can be found in this chapter. In the end of the chapter we briefly mention work that is related and present a running example which will be used in chapter 5 to test how useful our encodings are.

2.1 Security protocols

A communications protocol defines how parties should communicate. The parties in this thesis are generally two computers but can also be people or a person interacting with a computer.

In some cases the transport medium, also known as communication channel, cannot be trusted. For example, communication by post where somebody at the post-office opens and reads all your letters. Untrusted channels create the need of security protocols. These are communications protocols that are supposed to ensure certain properties in the presence of an adversary.

Using wax to seal a letter can be seen as part of an early security protocol. The seal provided both sender authentication and protection from modification.

The capabilities of the adversary depends on the circumstances. In some cases it is safe to assume that the adversary can only look at communication and in other cases one has to assume that the attacker may control one or more of the parties in the protocol. A protocol that uses a wax seal as authentication must assume that the adversary cannot create a forgery of an existing seal.

For communication between computers, cryptography may be used to achieve similar properties as the wax seal did for letters. We mainly focus on three primitives: nonces, hashes and encryption. Nonces are random values. These can be used for example to ensure that messages cannot be reused, or to avoid an attacker iden-

tifying if the same encrypted message was sent twice. Hash functions are one-way deterministic functions, in other words it is hard to identify what was hashed but one can use two hash values to check if they were generated from the same message. Encryption is used to hide the contents of a message while communicating.

In the next sections we describe properties that may be of interest in a security protocol, what properties a communication channel may have and some types of attacks an adversary can use against a protocol.

2.1.1 Properties of security protocols

There are multiple properties that may be required of a security protocol. We list the most common properties but note that there are more properties that could be required of a security protocol. We have for example omitted properties such as privacy and observational equivalence as they are not considered in this thesis and are hard to define in a compact form.

Secrecy: One of the most intuitive properties in a security protocol is secrecy. This property states that an adversary cannot learn something that was meant to be secret. This can for example be the key generated from a key exchange protocol.

Authentication: The basis of authentication is that the protocol ensures that the sender of a message is who she says she is. There are multiple degrees of authentication ranging from weak-authentication where one party only can be sure that the other party ran the protocol at some point, to injective authentication where you are sure the other party ran the protocol recently with you and conversely. For a more complete description of authentication, see [13].

Parameter freshness: The protocol guarantees that a parameter that is used, for example a key, has not been used before. This is important as many cryptographic primitives rely on the fact that a key is used for a limited amount of times.

2.1.2 Attack types

In this section we list the types of attacks on protocols that are described in the book 'Protocols for Authentication and Key Establishment' [14]. We ignore the *Certificate manipulation* attack as it focuses on public key encryption with certificates, something that is not applicable in our setting.

Eavesdropping: This is a passive attack where the adversary tries to gain knowledge by listening to messages without interfering. It is often used together with another attack such as replay. A trivial example is somebody listening to your phone call during a private conversation.

Modification: In this case the adversary takes parts of existing messages to craft another message that can be used to attack the protocol. For example if an adversary can change the amount of money in a message, this is a modification attack.

Replay: A replay attack is when the adversary may replay a previous message. For example re-using a coupon to get another discount. A special case of replay is *reflection* where the messages sent are returned to the sender, causing it to believe it is communicating with another valid party when it is actually communicating with itself.

Preplay: This is when the adversary runs the protocol before it was initialized by the parties. One can consider the more general case of message insertion where the adversary can insert a message at any time in a protocol. This can be used ito gain knowledge that can be used together with other attacks.

Typing attacks: A typing attack is when a message of one type is used as another type message. For example if part of a protocol decrypts an encrypted nonce without verifying what it decrypted, that part may be used to decrypt an encrypted secret instead.

Denial of service: This is when the adversary prevents the legitimate users from finishing a run of the protocol. A common example today is when an adversary floods a server with so many requests that it does not have time to deal with legitimate users. Another example is a thief cutting the phone line of a house that uses a dialling alarm system.

Cryptanalysis: The adversary may use knowledge about how the encryption is used in the protocol to break security properties. It could for example be the case that an encryption key could be calculated if the adversary knew both the encrypted message and its plain text content for multiple messages.

Protocol interaction: This is when two protocols interact to create a weakness. For example if the same encryption key is used for both protocols there may be a typing attack where a message from one protocol may be used in the other.

2.1.3 Channel types

Protocols use channels to communicate. As messages may be transferred by different means there are multiple channel types.

In [15] four types of channels are described including the *insecure channel*. If we have one sender Alice and one receiver Bob then in a *authentic channel* Bob can rely on the fact that the message was sent from Alice and intended for Bob. In a *confidential channel* Alice can rely on the fact that only Bob can read the message sent by Alice. The combination of these two channels creates a *secure*

channel. The fourth channel type is the *insecure channel*, which does not have any of these properties. We use the term *private channel* instead of *secure channel* when the channel does not use encryption but is secure because the adversary has no knowledge of the channel. We use the term *public channel* to describe an *insecure channel*. Cryptographic methods may be used on a public channel to gain properties of more secure channels.

Note that these properties do not mention protection from replay attacks. In the paper it is unclear if a message sent once can be received multiple times in any of the channels. In this thesis private channels are considered to have replay protection.

2.2 Formal verification of security protocols

Formal verification is done by assuming a model of a system and then using formal methods to prove properties of a protocol executed in this model.

The model that we assume is a variation of the Dolev-Yao model as described in detail in the next section. This model allows us to avoid difficulties with the probabilistic properties of encryption while still being meaningful to use for verification.

By using a protocol specification and the Dolev-Yao model one can attempt to prove properties of the protocol. Manual proofs of properties are usually long and error prone, therefore automatic verification tools are commonly used.

2.2.1 Dolev-Yao Model

The Dolev-Yao model is often described as a model where the adversary has control over all communication channels but cannot perform any attacks based on cryptanalysis. In this thesis we keep the notion of an adversary that can not attack the cryptographic primitives. We also weaken the adversary by not allowing her to interact with private channels.

The adversary can perform four actions as noted in [16], we present the list adapted to our notion of public channels.

- Read a message on a public channel and remove it from the channel.
- Split a message into smaller parts and remember them.
- Generate new data and combine parts into new data.
- Send messages generated from known data to a public channel.

From these actions it is clear to see that an adversary cannot use cryptanalysis in her attacks. For exampe, an attack that cannot be used by the adversary in the Dolev-Yao model is modification of an encrypted message that lacks integrity checks. In reality, if encrypted messages lack an integrity check, then it may be possible to modify single bits of the encrypted message to change the decrypted

message. Another example of an attack that does not exist in the Dolev-Yao model is brute-force. If a key is short or generated from a bad randomness source, the adversary still cannot brute-force the encryption key.

2.2.2 Undecidability and state space explosion

State space explosion is a well known problem for model checking. Protocol verification is no exception. As there may be many, even infinite, ways to execute a protocol, it is impossible to examine all paths of execution. Obviously, if protocol verification required the verifier to check all paths of execution, verification would be an undecidable problem for protocols with infinitely many traces.

Indeed it has been shown that protocol verification is undecidable in general [5, 6, 7]. Verification of secrecy is well studied and has been shown to be undecidable for protocols with, among other, limitations on the number of actors and message length [5].

As undecidability is a fact, no tool can verify all protocols. Generally this means that the tool may not terminate as it explores an infinite number of traces, or that it will give a result stating that it does not know.

2.2.3 Verification tools

There are numerous tools for verification of security protocols. We start off by describing bounded verification and one tool that uses bounded verification. We then continue to unbounded verification and some tools that perform unbounded verification.

In bounded verification a limit is set on certain parameters of the protocol. The protocol is considered secure if no attack is found within that bound. The bounds are generally set to avoid state space explosion. An example is setting a bound on the maximum length of a message.

The reason for using bounded verification is to achieve decidability and guarantee termination. If the property we want to prove is decidable in our bounded case, then we can guarantee termination.

Bounded verification does not guarantee that no attack exists as there may be an attack outside of the bound. In some cases this is not a problem. For example if the protocol specification itself has similar bounds. One argument for bounded verification is that the attacks that have been found on realistic protocols are generally small and possible to find with bounded verification [17].

As explained earlier we only mention one tool for bounded verification, namely SATMC [18]. This tool has been used in multiple projects, for example, AVISPA[1], AVANTSSAR[2] and SPaCIoS[3]. SATMC is mostly interesting as it has an expressive language that allows for models that closely resemble the implementation of a pro-

[1] http://www.avispa-project.org/
[2] http://www.avantssar.eu/
[3] http://www.spacios.eu/

tocol. It uses a bound on the number of protocol runs. The protocol is tested for attacks for an iteratively increasing number of rounds until the bound is reached.

Unbounded verification does not apply any limits when verifying a protocol. Instead tools for unbounded verification suffer from being hard to use. Tools for unbounded verification seldom guarantee termination as the general case of for example secrecy is undecidable. The tools assume that a property is decidable for the specific protocol we input, and tries to create a proof that the property holds. Even if the property is decidable the tool may not terminate. It can for example get stuck trying to find an attack by adding infinitely many participants to a protocol run.

To achieve a better probability of termination some tools use approximations. These approximations increase the capabilities of the adversary by for example relaxing the order in which operations in a protocol has to be executed.

An analogy can be drawn to the mortality problem, which is undecidable. The mortality problem is: "Given a Turning machine, decide whether it halts when run on any configuration". An approximation could be done by modifying any input machine by transforming all loops into infinite loops. This modified version will terminate in fewer cases as loops lead to non-termination. Because of this modification of the machine it may be easier to show if it will terminate. Thus, if we show termination for the modified machine, then termination for the original machine also holds. If we show non-termination of the modified version we cannot be sure if the original version terminates.

The tool targeted in this thesis is ProVerif [10]. It takes a protocol specification and internally converts it to a simpler representation on which it tries to verify the given properties. The translation is an approximation that results in false positives. We call false positives false attacks as they are proofs that an adversary can find e.g., the secret.

StatVerif [19] extends ProVerif to handle global state. StatVerif is discussed later in the related work section.

Tamarin [20] uses a backwards-search to prove the properties on a specification. Because Tamarin does not make any approximations, there are no problems with false attacks. Instead, the main issue with Tamarin is termination, which sometimes can be solved by guiding the proof manually.

In [21] an extension to Tamarin is introduced. The extension translates protocol specifications in the applied π-calculus into a model understood by Tamarin. This gives a more intuitive way to describe protocols for Tamarin. This translation is proven sound but may still require the user to guide Tamarin towards a proof.

2.3 Applied π-calculus

> *"[π-calculus] is an important result of the search for a calculus that could serve as the foundation for concurrent computation, in the same way in which the lambda calculus is a foundation for sequential computation." [22]*

A process calculus can be used to describe concurrent computation. In other words how programs communicate and interact with each other in an environment where several programs may execute simultaneously. The main construct in a process calculus is process parallelization, which can be seen as launching a new thread, and communication with other processes.

Applied π-calculus was originally defined in [8] and extends the π-calculus [23, 24]. Applied π-calculus adds symbolic application of functions and equations to the π-calculus.

A process in the applied π-calculus is equipped with a finite set $\mathcal{F}$ of functions and their arity, an infinite set of names $\mathcal{N}$, an infinite set of variables $\mathcal{V}$ and an equational theory Σ.

Terms can be created from names, variables and functions. The equational theory maps relations between terms. Symbolic application of functions means that if we have a term t and a function f of arity one, the application of f on t results is the term $f(t)$. A function without any equations can be considered to be one-way hash functions. They do not disclose their arguments but generate a random value which is deterministic for the input.

To model a function that is not a hash function, a destructor has to be defined. The destructor is defined by adding equations to the equational theory. If for example $(g(f(a)) = a) \in \Sigma$, then it is possible to get back the t from the term $f(t)$ by creating the term $g(f(t))$ which, modulo the equational theory equals t. The function g is a destructor for the function f. Another example of a destructor is a decryption function.

A process in the applied π-calculus is a sequence of operations. All operations can be seen in the applied π-calculus grammar in figure 2.1. The null process is a process that does nothing. Parallel composition runs two processes as different threads and replication describes a process that is run in parallel with itself an unlimited number of times. Name restriction binds a variable in the process to a new name. This can be viewed as a process creating a new random value or channel.

Lastly we have included the event operation from ProVerif. This operation does not affect the process but is added to the process trace. The process trace is a sequence of all executed operations. This trace can for example be used to show that it is possible to reach a specific point in a process, or used as part of a counter proof of secrecy.

$M, N ::=$	Terms
a, b, c	Names
x, y, z	Variables
$f(M_1, M_2, ...)$	Function application

$P, Q, R ::=$	Processes
0	Null process
$P \mid Q$	Parallel Composition
$!P$	Replication
$\nu n.P$	Name restriction
$if\ M = N\ then\ P\ else\ Q$	Conditional
$c(x).P$	Message input on channel c
$c(=x).P$	Syntax sugar for $c(x').if\ x = x'\ then\ P\ else\ 0$
$\overline{c}\langle M\rangle.P$	Message output on channel c
$event(e(M))$	Log event e(M) in trace

Figure 2.1. The grammar of processes in applied π-calculus.

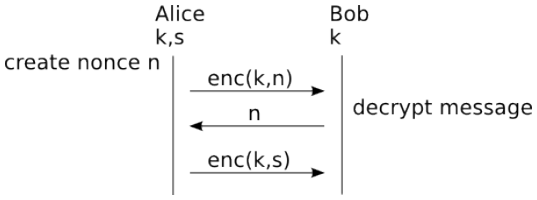

Figure 2.2. A trivial protocol described in Alice and Bob notation.

2.3.1 Example

In figure 2.2 the message sequence chart of a simple protocol is presented. Alice sends a nonce encrypted with the key k to Bob, Bob replies with the decrypted nonce as proof of knowing the encryption key and then Alice sends the secret encrypted with the same key.

Figure 2.3 shows the same protocol described in applied π-calculus. The process has one equation $(dec(k, enc(k, m)) = m) \in \Sigma$. The description in applied π-calculus is more specific than the message sequence chart. It specifies that the communication between Alice and Bob can be intercepted by the adversary and also that the protocol may run multiple times for the same k and s. This is not clear from the message sequence chart.

In this protocol an attacker can get the key by observing one interaction of Alice and Bob to get the encrypted secret. The attacker can then initialize a session with Bob and send the message containing the encrypted secret as the message that Bob

$$A(k,c) \triangleq \nu s.\nu n.\overline{c}\langle enc(k,n)\rangle.c\langle =n\rangle.\overline{c}\langle enc(k,s)\rangle.0$$
$$B(k,c) \triangleq c\langle m\rangle.\overline{c}\langle dec(k,m)\rangle.c\langle m_2\rangle.0$$
$$Main \triangleq \nu k.\nu c.\overline{pub}\langle c\rangle.!(A(k,c)|B(k,c))$$

Figure 2.3. Protocol specification as described in 2.3 and depicted in figure 2.2. s is the pre-shared secret, k is a pre-shared key and c is the public communication channel.

$$(\mathcal{E},\mathcal{A},\mathcal{P}\cup\{0\}) \rightarrow (\mathcal{E},\mathcal{A},\mathcal{P}) \quad (1)$$
$$(\mathcal{E},\mathcal{A},\mathcal{P}\cup\{P|Q\}) \rightarrow (\mathcal{E},\mathcal{A},\mathcal{P}\cup\{P\}\cup\{Q\}) \quad (2)$$
$$(\mathcal{E},\mathcal{A},\mathcal{P}\cup\{!P\}) \rightarrow (\mathcal{E},\mathcal{A},\mathcal{P}\cup\{P\}\cup\{!P\}) \quad (3)$$
$$(\mathcal{E},\mathcal{A},\mathcal{P}\cup\{\nu n.P\}) \rightarrow (\mathcal{E}\cup n',\mathcal{A},\mathcal{P}\cup\{P\{^{n'}/_{n}\}\}) \text{ if } n'\notin\mathcal{E} \quad (4)$$
$$(\mathcal{E},\mathcal{A},\mathcal{P}\cup\{\overline{c}(M).P\}) \rightarrow (\mathcal{E},\mathcal{A}\cup M,\mathcal{P}\cup\{P\}) \text{ if } c\in\mathcal{A} \quad (5)$$
$$(\mathcal{E},\mathcal{A},\mathcal{P}\cup\{c(x).P\}) \rightarrow (\mathcal{E},\mathcal{A},\mathcal{P}\cup\{P\{^{M}/_{x}\}\}) \text{ if } c,M\in\mathcal{A} \quad (6)$$
$$(\mathcal{E},\mathcal{A},\mathcal{P}\cup\{\overline{c}(M).P_1\}\cup\{c(x).P_2\}) \rightarrow (\mathcal{E},\mathcal{A},\mathcal{P}\cup\{P_1\}\cup\{P_2\{^{M}/_{x}\}\}) \text{ if } c\notin\mathcal{A} \quad (7)$$
$$(\mathcal{E},\mathcal{A},\mathcal{P}\cup\{\textit{if } M=N \textit{ then } P \textit{ else } Q\}) \rightarrow (\mathcal{E},\mathcal{A},\mathcal{P}\cup\{P\}) \text{ if } M=_{\Sigma} N \quad (8)$$
$$(\mathcal{E},\mathcal{A},\mathcal{P}\cup\{\textit{if } M=N \textit{ then } P \textit{ else } Q\}) \rightarrow (\mathcal{E},\mathcal{A},\mathcal{P}\cup\{Q\}) \text{ if } M\neq_{\Sigma} N \quad (9)$$
$$(\mathcal{E},\mathcal{A},\mathcal{P}\cup\{\textit{event(e(M)).P}\}) \rightarrow (\mathcal{E},\mathcal{A},\mathcal{P}\cup\{P\}) \quad (10)$$

Figure 2.4. Operational semantics of the applied π-calculus.

needs to decrypt to prove that he knows the key.

2.3.2 Operational Semantics

A process is executed by following the rules given in the operational semantics in figure 2.4. Each rule has one initial process configuration and one resulting process configuration. In some cases there is also a side-condition on when the rule can be used. A process configuration is described by $(\mathcal{E},\mathcal{A},\mathcal{P})$. $\mathcal{E}$ is a set of all the names bound by the ν operation. νx binds the variable x to a unique new name. $\mathcal{A}$ is a set of terms. This represents the adversary knowledge, this is what the adversary knows or can generate by either creating new terms or applying equations to the terms she has. Lastly $\mathcal{P}$ is a multiset of processes where all processes are executed in parallel.

We have chosen to include the adversary knowledge in the operational semantics to specify when an output or input operation can be executed and how the adversary can affect the process. If a term known by the adversary is used for communication this is considered a public channel. If the term is not known by the adversary, it is a private channel. If a process outputs on a private channel, there is a requirement

$(\{pub\},\{pub\},$	$\{\nu c.(\nu s.\overline{c}\langle s\rangle.0 \mid c\langle x\rangle.\overline{pub}\langle x\rangle.0)\})$	Initial conf.
$(\{pub\} \cup \{c'\},\{pub\},$	$\{(\nu s.\overline{c'}\langle s\rangle.0 \mid c'\langle x\rangle.\overline{pub}\langle x\rangle.0)\})$	Rule 4 applied
$(\{pub\} \cup \{c'\},\{pub\},$	$\{\nu s.\overline{c'}\langle s\rangle.0\} \cup \{c'\langle x\rangle.\overline{pub}\langle x\rangle.0\})$	Rule 2 applied
$(\{pub\} \cup \{c'\} \cup \{s'\},\{pub\},$	$\{\overline{c'}\langle s'\rangle.0\} \cup \{c'\langle x\rangle.\overline{pub}\langle x\rangle.0\})$	Rule 4 applied
$(\{pub\} \cup \{c'\} \cup \{s'\},\{pub\},$	$\{0\} \cup \{\overline{pub}\langle s'\rangle.0\})$	Rule 7 applied
$(\{pub\} \cup \{c'\} \cup \{s'\},\{pub\},$	$\{\overline{pub}\langle s'\rangle.0\})$	Rule 1 applied
$(\{pub\} \cup \{c'\} \cup \{s'\},\{pub; s'\},$	$\{0\})$	Rule 5 applied
$(\{pub\} \cup \{c'\} \cup \{s'\},\{pub; s'\},$	$\{\})$	Rule 1 applied

Figure 2.5. Example execution of process using the operational semantics. The process outputs a secret on a private channel, inputs it in a parallel subprocess and outputs it to the adversary.

that there must be another process with an input operation on that channel in order to let the processes continue. This is not required for output and input on a public channel where all communication can be viewed as direct communication with the attacker.

In this thesis we consider the initial process configuration to always contain a public name 'pub' which can be used to explicitly communicate information to the adversary such as new terms that are supposed to be public. More formally, the initial configuration is $(\mathcal{E}, \mathcal{A}, \mathcal{P})$ where $pub \in \mathcal{E}$ and $pub \in \mathcal{A}$.

A protocol run trace is a list of the applied rules and the initial state. This list details how the protocol was executed and when every message was sent. A trace can for example be used to show that an attack exists or that a message can be sent.

An example of the process $\nu c.(\nu s.\overline{c}\langle s\rangle.0 \mid c\langle x\rangle.\overline{pub}\langle x\rangle.0)$ executed using the operational semantics can be seen in figure 2.5. The first line depicts the initial configuration of our process. For each new line one of the rules from the operational semantics has been applied. In the end of this execution we see that the adversary has knowledge of the name s'. Note that while this example process only has one way of execution, most processes can be executed in several ways by applying the rules in different order.

2.3.3 Extending applied π-calculus with state

In [25] the applied π-calculus is extended with state. In this section we describe an extended version of the applied π-calculus based on their work. Figure 2.6 describes the addition to the applied π-calculus grammar. As the applied π-calculus is made for parallel processes, locks are also added to the grammar. Locks allow processes

$P, Q, R ::=$	Processes
$[s \to M]$	Initialize state s with M
$s := M.P$	Write M to state cell s
read s *as* $x.P$	Read state cell s
lock $s.P$	Lock state cell s
unlock $s.P$	Unlock state cell s

Figure 2.6. Addition to the applied π-calculus grammar to support state.

$$(\mathcal{S}, \mathcal{E} \cup s, \mathcal{A}, \mathcal{P} \cup \{([s \to M].0, \mathcal{L})\}) \to (\mathcal{S} \cup \{s \to M\}, \mathcal{E} \cup s, \mathcal{A}, \mathcal{P}) \quad \text{if } \{s \to N\} \notin \mathcal{S} \tag{11}$$

$$(\mathcal{S} \cup \{s \to M\}, \mathcal{E}, \mathcal{A}, \mathcal{P} \cup \{(lock\ s.P, \mathcal{L})\}) \to (\mathcal{S} \cup \{s \to M\}, \mathcal{E}, \mathcal{A}, \mathcal{P} \cup \{(P, \mathcal{L} \cup s)\}) \quad \text{if } s \notin (\mathcal{L} \cup locks(\mathcal{P})) \tag{12}$$

$$(\mathcal{S} \cup \{s \to M\}, \mathcal{E}, \mathcal{A}, \mathcal{P} \cup \{(unlock\ s.P, \mathcal{L} \cup s)\}) \to (\mathcal{S} \cup \{s \to M\}, \mathcal{E}, \mathcal{A}, \mathcal{P} \cup \{(P, \mathcal{L})\}) \tag{13}$$

$$(\mathcal{S} \cup \{s \to M\}, \mathcal{E}, \mathcal{A}, \mathcal{P} \cup \{(read\ s\ as\ x.P, \mathcal{L})\}) \to (\mathcal{S} \cup \{s \to M\}, \mathcal{E}, \mathcal{A}, \mathcal{P} \cup \{(P\{^{M}/_{x}\}, \mathcal{L})\}) \quad \text{if } s \notin locks(\mathcal{P}) \tag{14}$$

$$(\mathcal{S} \cup \{s \to M\}, \mathcal{E}, \mathcal{A}, \mathcal{P} \cup \{(s := N.P, \mathcal{L})\}) \to (\mathcal{S} \cup \{s \to N\}, \mathcal{E}, \mathcal{A}, \mathcal{P} \cup \{(P, \mathcal{L})\}) \quad \text{if } s \notin locks(\mathcal{P}) \tag{15}$$

Figure 2.7. Additional rules to the operational semantics.

to ensure that no other process is modifying the state while the lock is held.

With the addition of state to the grammar, we also must modify the operational semantics to handle the new operations. To handle the global state cells, a state term is added to the process configuration resulting in the tuple $(\mathcal{S}, \mathcal{E}, \mathcal{A}, \mathcal{P})$. To handle the locking mechanisms that are allowed by the grammar, each process $\mathcal{P}$ in the process configuration is made into a tuple $(P, \mathcal{L})$. In this new tuple P is the process as described earlier and $\mathcal{L}$ is the set of locks held by this process. $\mathcal{S}$ is a set that maps state cells to a value, $\{s \to M\}$ indicates that state cell s has value M.

The extension in [25] has three additional requirements for processes with state. We do not add these requirements as we deem them unnecessary for handling state. Later when encoding state in the original applied π-calculus we add requirements to the process in order to use the encoding defined in that paper.

Note that the operational semantics in figure 2.7 only describes private state cells that are not available to the adversary. A public state cell is a cell, which the adversary knows the name of. An adversary can do the same operations as a process on a public state cell, thus only read or write the value of a public state cell when it is not locked by any process.

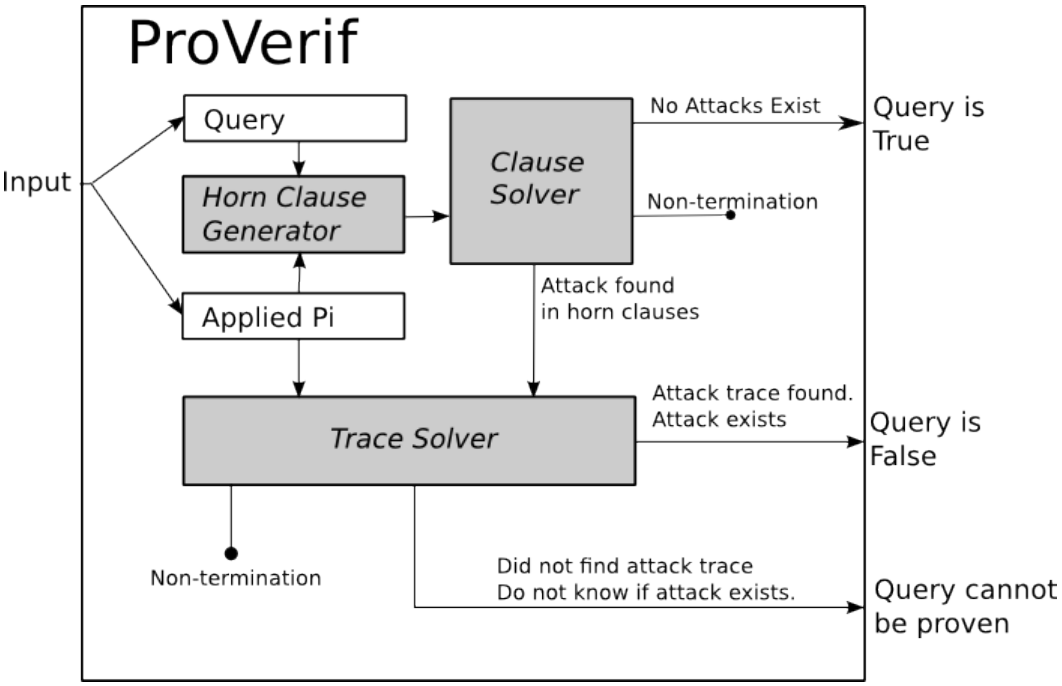

Figure 2.8. The internals of ProVerif. A 'Cannot be proven' output indicates a false attack in the Horn clauses.

2.4 ProVerif

ProVerif is a tool for automatic verification of security protocols. It uses the applied π-calculus as input language and internally generates Horn clauses to verify a property. ProVerif has been used and developed since 2001 and is well documented.

We describe how Horn clauses are used to verify protocol properties, discuss false attacks, and mention possible causes of non-termination. To help the reader through this section a rough visualization of the internals of ProVerif can be seen in figure 2.8. The input is translated to Horn clauses, which are then used to prove properties on the protocol. If an attack is found in the Horn clause representation, ProVerif attempts to find a valid attack trace in the original process description. This is done by attempting to go backwards from the Horn clause solution. If a trace is found the attack is valid and steps to reproduce the attack are output. Finally, if no trace is found, "cannot be proved" is output. This is a "don't know" answer.

2.4.1 ProVerif input

A protocol is described by one main process, functions, equations, and, queries. ProVerif also allows the user to describe predicates with Horn clauses to gain additional control over the translation.

$$\neg A \vee \neg B \vee C \equiv A \wedge B \rightarrow C$$

Figure 2.9. Horn clauses can be viewed as implications

Queries specify the properties that we want ProVerif to prove, they can for example consist of checks for adversary knowledge or event reachability. These are central to the verification as they define what is to be proven. A badly formulated query may be verified even though there are attacks in the protocol.

Equations describe the relation of functions. One common usage is to describe the exponentiation where a function exp(n,m) describes $n^m \ mod \ p$. The relation $g^{xy} \ mod \ p = g^{yx} \ mod \ p$ can be described by the equation $exp(exp(g,x),y) = exp(exp(g,y),x)$

Functions are categorized into two types: constructors and destructors. Unless a destructor is specified a function can be considered as a one-way hash function. As noted in the applied π-calculus section, a destructor is a function that extracts a term by the use of equations. A destructor specifies how values can be extracted after a function has been applied. The equation $dec(k, enc(k,m)) = m$ describes decryption, the *dec* function is in this case a destructor.

Events can be viewed as simple log messages that will be inserted into the protocols run trace without any impact on the protocol. These are often used as a debugging tool to show that a protocol can complete a session. Later in the thesis we show how these can be used in order to prove freshness in a protocol.

The predicates that are allowed by ProVerif lets the user define lists and notions such as random selection from a list. Predicates are later used to avoid false attacks while encoding state.

2.4.2 Horn clauses

A Horn clause is a set of predicates combined by the logical *or* operation, where at most one predicate is non-negated. This gives us three types of Horn clauses. **Facts** consist of a single non-negated predicate. **Rules** have more than one negated and one non-negated predicate. Lastly **Goals** are negative clauses with no positive predicate. The different types of Horn clauses are named after how they can be used during resolution. Rules can be viewed as implications as seen in figure 2.9. These can be used together with facts to gain knowledge. A goal clause is a clause that we are trying to prove to be true.

ProVerif creates multiple rule and fact clauses whose predicates contain arguments, then it tries to reach the goal using these clauses. The clauses are mainly based on adversary knowledge. Predicates named 'attacker' have a single argument which signifies one term that the adversary knows.

A simplified view of the clauses used to derive the fact that the adversary can

```
Clause 1: attacker(enc(key,n[!1 = @sid_90]))
Clause 2: attacker(n[!1 = @sid_106]) -> attacker(enc(key,secret))
Clause 3: attacker(enc(key,nA_132)) -> attacker(nA_132)
```

Figure 2.10. Example of clauses used to derive the fact that the secrecy of the term 'secret' does not hold in figure 2.3.

get hold of the secret in the process from figure 2.3 can be found in figure 2.10. Variables used as arguments in the predicates may have arguments of their own. These arguments are not used during the Horn clause resolution of ProVerif, they are instead used by ProVerif when trying to construct a real attack on the protocol. Nested arguments are for example used in nonces to specify in which process the nonce was created.

Clause 1 describes that the attacker knows a message with an encrypted nonce. Clause 2 describes that if the adversary knows a nonce generated by Alice he can get the encrypted secret. Clause 3 describes that if the adversary knows an encrypted message he can get the decrypted message. Note that n[!1 = @sid_90] specifies the fact that nonce n has the session variable sid_90 as argument. This argument is used by ProVerif to identify that the n used in Clause 2 was generated from Clause 1.

In this case we see that the adversary can use Clause 1 to get an encrypted n, then Clause 3 to get the unencrypted version. From this she can use Clause 2 to get the encrypted secret and lastly use Clause 3 to get the secret. One can identify that Clause 1 and 2 are generated from the Alice role while Clause 3 is generated from the Bob role.

2.4.3 False attacks

It is important to note that while the translation to the Horn clause representation is an approximation, the approximation is sound: if an attack exists in the original model, it also exists in the approximation, but if an attack is found in the approximation it is not guaranteed that it exists in the original model.

In figure 2.8 we can see how ProVerif tries to verify a query. When ProVerif finds an attack in the Horn clause representation, it tries to construct the attack in the modelled protocol. If it succeeds a true attack is found and it outputs true or false. If ProVerif finds an attack in Horn clause representation but fails to reproduce it in the model, then it outputs 'cannot be proven'. This is what we call a false attack. We have identified two causes of false attacks. Both are a result of the fact that the generated Horn clauses always are considered to be true. In other words, if two operations are mutually exclusive, both can be performed in the Horn clause representation. There is no way of removing the second operation from the Horn clauses after the first one has been performed.

The first one appears when using predicates in ProVerif. If we have a predicate

describing *take one element at random from list A*, this can in the Horn clause representation be understood as *launch a new parallel process for all elements in list A*.

The other one is a well documented false attack [10, section 6.3.4]. In a process where something is input and later a secret output, the Horn clause approximation allows an adversary to input the secret before it is output. To clarify, given a process P=$\nu k.c\langle n\rangle.\bar{c}\langle k\rangle$ with a public channel c, the clauses generated by ProVerif for this process allows the adversary to input k at $c\langle n\rangle$ although the adversary has no knowledge of k yet.

2.4.4 Termination

A model in ProVerif can give rise to two types of termination problems. The application of the Horn clauses can generate an infinite loop leading to ProVerif not terminating. If we have non-termination due to this, we have no knowledge about what result ProVerif would output if it were to terminate.

The other cause for non-termination is unification. When ProVerif has found a trace in the Horn clause representation, it needs to verify that this corresponds to a trace in the model. This can take a long time and much memory. If we reach this problem then we know that ProVerif either finds a trace or outputs 'cannot be proven'. It never returns the result that no attack exists. We are not sure if this latter cause of non-termination is a true case of non-termination or if the algorithm would eventually finish, given enough time.

2.5 Related work

The ProVerif manual [10, section 6.3] describes useful constructs for modelling a protocol, especially for verification in ProVerif. There has been some work done to avoid false attacks in ProVerif namely [26] which describes a method to avoid some of them by using phases for verification of commitment protocols.

As mentioned in the tools section, StatVerif [19] is an extension to ProVerif which adds support for global state. It does so by providing new translations from state operations to Horn clauses. One limitation is that it does so in an old version of ProVerif and with an old input language. The StatVerif paper briefly mentions encoding state by using private channels but does not elaborate on the implementation and actual limitations.

[27] goes in the other direction and formalizes a way to remove aspects of a protocol that is not needed for verification to reduce the size of the model and make verification faster.

We have not found any papers describing how to verify freshness in ProVerif.

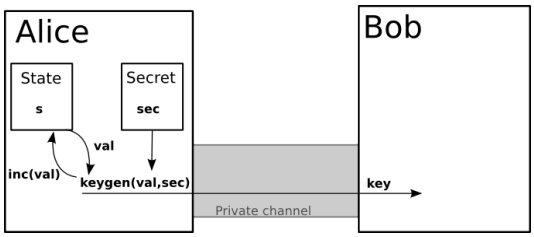

Figure 2.11. MiniDC. Alice sends a value generated from its counter and a secret over a private channel to bob. Bob verifies that the received value is fresh.

2.6 Running example: MiniDC

We define a protocol named MiniDC which we use as a running example for the rest of the thesis. Figure 2.11 depicts the protocol. Alice has a local counter s and a secret *sec*. Alice has a loop that increases the counter and generates a key from the secret and the counter value. This generated key is sent to Bob over a private channel. We want to verify that Bob will never receive the same key twice.

The protocol is loosely based on the Dual Connectivity protocol mentioned in section 1.1.

As we so far in the thesis have not introduced any way of modelling freshness in the applied π-calculus, we use brackets to describe freshness.

MiniDC described in our extended applied π-calculus can be found in figure 2.12. The first four lines create constants used. The constants are the private channel c, a value that represents the number zero, Alices secret *sec* and Alices state cell s. The fifth line initializes the state cell s to the value *zero*. The replicated process on lines 7-11 describes a single instance of Alice. The lock and unlock operation force each of the replicated processes not to run in parallel while still allowing for an infinite number of counter increments.

Line 9 sends the term $keygen(val, sec)$ over the channel c. After this the counter in val is increased and stored.

Lines 13-14 describe the replicated Bob process. Bob simply receives a key on the channel c. Line 14 indicates that we want to verify that the same key cannot be received twice on line 13.

Although not necessary we define an equation and let $\Sigma = \{dec(inc(x)) = x\}$. This equation allows a process or adversary not only to increase values, but also to decrease them.

As we only have a single state we can see that the same counter value will never be read twice in two different replications of Alice, thus the generated key is always fresh.

As we communicate over a private channel there are no kinds of attacks, thus the key received by Bob is always fresh. It is trivial to see that secrecy of all the generated keys holds because there is no communication on public channels and the adversary does not know *sec*.

Line	Operation	Description
1	$\nu c.$	Create private channel c
2	$\nu zero.$	Create constant zero
3	$\nu sec.$	Create secret sec
4	$\nu s.$	Create state cell s
5	$[s \rightarrow zero]\vert$	Initialize state
6	!((	Replicate Alice process to achieve loop
7	$lock\ s.$	
8	$read\ s\ as\ val.$	Read counter from state
9	$\overline{c}\langle keygen(val, sec)\rangle.$	Generate key and sent do Bob
10	$s := inc(val).$	Store increased counter
11	$unlock\ s$)	
12	|(	Replicated Bob process
13	$c\langle key\rangle.$	Receive key
14	*{verify that key is fresh}*	
15	))	

Figure 2.12. Description of MiniDC in our extended applied π-calculus. A loop is incrementing a counter and generating keys while a parallel process is receiving the keys.

Chapter 3

Encoding state and freshness in the applied π-calculus

ProVerif has no native support for verification of event freshness nor the applied π-calculus extended with state. To verify freshness in a stateful protocol with ProVerif we must be able to encode it in the non-extended applied π-calculus. We must also encode freshness in a way that ProVerif is able to verify.

In this chapter we define an encoding for state in the applied π-calculus. We also define a method that can be used to verify freshness of events in tools that are able to verify secrecy.

These encodings are later used to verify freshness of the keys generated by Alice in MiniDC.

3.1 State in applied π-calculus

In [25], private state cells are encoded by input and output on a private channel. In the encoding, the state is always owned by some process but can be handed over to some other process through the private channel.

We describe an encoding that is equivalent to their but has a clearer connection between the process described in applied π-calculus with state and the same process encoded in the non-extended applied π-calculus.

Some processes defined in the extended applied π-calculus from section 2.3.3 can be encoded in the regular applied π-calculus. To ease the encoding we require that any process that reads or locks a state cell s must do so by the operation sequence *lock* s.*read* s *as* $x.P$. Also, we require a that process which writes or unlocks a state cell s must do so by the operation sequence $s := M.$*unlock* $s.P$. In other words, a process may not lock a state without reading it and may not unlock a state without writing to it. Note that the process may write the current state value back to the state cell.

Any state process may be converted into this form if all reads and writes of a locked state are considered to be done to local variables where the last local value

is written to the state cell during unlock. As *read* s *as* $x.P$ or $s := x.P$ outside a locked process only is possible if the state is not locked, we can describe the former by *lock* s.*read* s *as* $x.s := x.$*unlock* $s.P$ and the latter by *lock* s.*read* s *as* $x'.s := x.$*unlock* $s.P$. By doing this, reading or writing while not locked is no longer an atomic operation.

We assume that the fact that unlocked read and unlocked write becomes non-atomic operations does not affect verification of properties in ProVerif. Our argument for this is that there is no fail condition when using a locked state. Time dependent properties may be affected by the change to non-atomic operations, but as ProVerif does not support verification of time dependent properties this is not an issue.

Apart from the requirement of operation sequence mentioned above we define some additional requirements needed by our state encoding. These requirements ensure that the encoded process still behave as the original process described in the extended applied π-calculus. The added requirements are listed below.

1. There may be no replication or parallelization between lock and unlock of the same state.

2. It shall only be possible to initialize a state once.

3. The name of a state cell may not be used as a channel.

4. The name of a state cell may not become public.

If these requirements are not present, encoding the process would not match the operational semantics defined for applied π-calculus with state. For example, according to the operational semantics the rule to initialize a state cell can only be applied once for each state cell. The second requirement enforces this, allowing us to skip this check in our encoding.

Requirement 4 forces the state cell to be private.

The intuition behind the encoding can be understood by the analogy with a talking stick. A talking stick is an item used during group discussions to signal who has the word. The holder of the stick is the only one allowed to speak. We will consider the state cell to be the talking stick. Holding the stick is analogous to have a lock on the state cell, putting the stick on a table is the act of unlocking the state cell. If you hold the stick you can look at it: read the state cell. You can also modify it: assign a new value to the state cell.

We encode this by creating a channel for each state cell. This channel is the table in the analogy. The state value is a message that can be read from the channel. Locking and reading the state cell is done by reading the value from the channel. Writing and unlocking is done by sending a message on the channel. The translations can be seen in figure 3.1.

Note that if the state cell becomes public, this encoding does not hold, we deal with this in the next section.

$$[s \to M].0 \Rightarrow \overline{s}\langle M\rangle.0$$
$$lock\ s.read\ s\ as\ x.P \Rightarrow s\langle x\rangle.P$$
$$s := M.unlock\ s.P \Rightarrow \overline{s}\langle M\rangle.P$$

Figure 3.1. Encoding of state in the applied π-calculus

Operation	**Description**
$\nu c_f.$	Create freshness channel c_f
$\nu s_f.$	Create freshness secret s_f
$\nu c.$	Create private channel c
$\nu zero.$	Create constant zero
$\nu s.$	Create secret s
$\nu c_s.$	Create state cell c_s
$\overline{c_s}\langle zero\rangle \mid$	Initialize state
!((	Replicated Alice process
$\quad c_s\langle val\rangle.$	Lock+Read state into val
$\quad \overline{c}\langle keygen(val, s)\rangle.$	Generate key and sent do Bob
$\quad \overline{c_s}\langle inc(val)\rangle)$	Write+Unlock state with inc(val)
$\mid($	Replicated Bob process
$\quad c\langle key\rangle.$	Receive key
$\quad \nu n_f$	Create nonce used for freshness
$\quad (\overline{c_f}\langle n_f, key\rangle \mid$	Output key to freshness channel
$\quad c_f\langle n'_f, {=}key\rangle.$	Receive key from freshness channel
$\quad \underline{if\ n' = n\ then\ 0\ else}$	Stop if message was produced by ourselves
$\quad \overline{pub}\langle s_f\rangle)$	Output freshness secret
))	

Figure 3.2. MiniDC where encodings are applied

3.1.1 Public state cells

Encoding state cells that may become public is done with the same replacements as for private. We remove the fourth requirement "The name of a state cell may not become public." and add two new.

4. The name of a state cell s must become public only by the operation $\overline{pub}\langle s\rangle$
5. The adversary may not get any other advantage than access to the state cell from learning the name of a state cell.

The reason we have the last requirement is that we will give the adversary a separate channel, which the adversary can use to perform operations on the channel that encodes the state cell. If for example the name of the state cell was used as a key, the stateful process would disclose the key when disclosing the cell name. The encoded process will output another name, and thus the adversary does not get the key.

The fourth requirement makes it easy to locate where we need to apply the encoding.

We note that it may be hard to prove that a process follows the last requirement. We assume that if a process only uses a name as a state cell, and any term derived

$$\overline{pub}\langle s\rangle.P \Rightarrow (\nu s_a.\overline{pub}\langle s_a\rangle.!s_a\langle i\rangle.s\langle n\rangle.\overline{s_c}\langle n\rangle.s_c\langle n'\rangle.\overline{s}\langle n'\rangle)|P$$

Figure 3.3. Encoding disclosure of private state cell

$$P_{main} \Rightarrow \nu f.\nu s.P_{main}$$
$$event(M).Q \Rightarrow$$
$$(event(e(M)).Q)|\nu n_1.(\overline{f}\langle M, n_1\rangle|f\langle =M, n_2\rangle.if\ n_1 = n_2\ then\ 0\ else\ \overline{pub}\langle s\rangle)$$

Figure 3.4. Encoding event freshness with secrecy of term s.

from that name is never learnt by the adversary, then she does not gain any other advantage.

The strictness of these requirements make the encoding, which can be seen in figure 3.3, simple. Where the private state cell was to be disclosed we instead create a new channel s_a which for the adversary will work as a proxy for the state, enforcing the rules of locking and unlocking. The adversary can then send an initial message to s_a. This locks the state and outputs its value. This corresponds to a lock and read combination. Then the state will be locked by the adversary until she outputs another message on s_a which is the value written to the state channel, this corresponds to the write and unlock operation. The replication in the encoding allows the adversary to lock and unlock the state infinitely many times.

3.2 Encoding freshness with secrecy

Parameter freshness as described earlier is broad and hard to define as it does not relate to any specific point in the protocol. When analysing freshness it is common to look at a specific point of key generation and argue that the key has not been generated before. In this part we consider a related type of freshness, event freshness.

Definition 1. *Event freshness holds iff there exists no trace where two events* event(M) *and* event(N) *with* $M =_\Sigma N$ *are executed.*

This can be limited to subsets of M. For example if one is interested only in freshness for $event(keygen(N))$, then it is enough to apply the freshness encoding for events where it is possible for $M =_\Sigma keygen(N)$.

If we want to encode event freshness for events in process P_{main} we need to apply two translations to the process. These can be seen in figure 3.4. The first translation adds two global names, one secret s and one private channel f.

As ProVerif is able to verify secrecy, we encode event freshness as secrecy. The goal is that if secrecy for term s holds then event freshness also holds.

Like earlier we have the public channel *pub* that is used to give the adversary information. The second translation rule in 3.4 breaks secrecy of s if event freshness does not hold. This is done by three parallel processes. The first executes the event as before the translation was applied, the second generates a new nonce n_1 and outputs it on f together with the term that we want to verify as fresh. The third parallel process inputs a message on f and checks if the term came from our own output. If it did, then we do nothing but otherwise we output the secret to the adversary.

This encodes event freshness with secrecy of term s. If event freshness fails, then it is possible for the adversary to gain knowledge of s. If event freshness does not fail, then it is impossible for the adversary to know s.

There is an important distinction to make. What we have described is that if event freshness fails in one trace, there exists a trace where secrecy of s fails. It does not mean that if event freshness fails in one trace, then secrecy of s fails in that specific trace. Even though event freshness fails, the processes that fail freshness could, in that specific trace, input their own output and get stuck in the null operation. In figure 3.2 we have encoded event freshness in MiniDC. To prove event freshness we need to prove secrecy of the term s'

It is worth mentioning that there are multiple encodings that could be used for event freshness. We chose the encoding mentioned above as we feel it strikes the best balance between simplicity and possibility to use in ProVerif.

Another method of encoding freshness is for example to keep the replacement for P_{main} but the event is instead replaced with $event(M).Q|\overline{f}\langle M\rangle|f\langle =M\rangle.f\langle =M\rangle.\overline{pub}\langle s\rangle$. The thought behind this encoding is that each event produces one output but requires two inputs to break secrecy. Thus if we have two equal outputs, freshness has failed and our encoding outputs the secret s. In ProVerif this encoding would not work. The Horn clause representation allows the single output operation to be used for both input operations.

At the beginning of our work we identified another encoding of freshness that involved a public channel to avoid blocking output, and arguing for the knowledge of the adversary. That encoding is much more verbose than the one we have described in this section, and requires more new terms and equations. Therefore we have chosen to discard it even though we managed to use it in ProVerif.

Chapter 4

Encoding state and freshness in ProVerif

In this chapter we show how the encodings described in chapter 3 can be used in ProVerif. The encoding for state causes false attacks when verifying freshness in ProVerif. To mitigate this issue we identify several ways to avoid false attacks that appear when verifying freshness in ProVerif. We also describe how some of these methods can be applied to avoid false attacks caused by the state encoding. In chapter 5 we show how MiniDC is verified using ProVerif and these methods.

A false attack is caused by the abstractions of ProVerif. It is always in relation to a protocol property. If a false attack exists when verifying freshness, this false attack may not be a problem when verifying secrecy of some term. For example if a protocol always communicates on private channels, the abstractions of ProVerif may allow the same Horn clause to be applied multiple times, resulting in a false attack of freshness. This does not affect secrecy as all communication is done on private channels.

We only focus on false attacks that appear when verifying freshness, but we note that our methods may be applicable to avoid false attacks when verifying other properties.

To avoid all false attacks caused by the encoding of state a method that requires large modifications of the protocol has to be applied. It is possible to avoid some false attacks with a less intrusive method, which may be enough in some cases. We want to emphasise that although false attacks exist, the encoding may still find a true attack. That is, if the protocol has an attack there is a chance that ProVerif finds the real attack instead of the false attack for the protocol. Thus, the encoding can be used as a debugging tool.

There are no false attacks in our encoding of freshness, but ProVerif makes abstractions that are not well suited for freshness. If the term we want to verify freshness for is not trivially fresh, this usually leads to false attacks. By trivial we mean cases where we are verifying freshness for a new name and the name is not used in any later event. A common cause of false attacks on freshness is communication

over a channel. As the abstractions of ProVerif allows a single message to be sent multiple times, this may break freshness.

The first section describes how to avoid general false attacks. The second section discusses how these can be used for our encodings.

4.1 Avoiding false attacks in ProVerif

In this section we describe two ways of using nonces to avoid false attacks. Whenever ProVerif finds an attack that exists due to the abstraction to Horn clauses, it gives a 'cannot be proved' result. We want to avoid such cases to increase the chance of getting an actual result.

The first method we describe avoids false attacks caused by input and output on private channels, while the second method avoids false attacks caused by output from a process being used earlier in the same process. Lastly we describe a complex method that can be used to avoid all false attacks that appear when verifying freshness.

4.1.1 Avoiding false attacks due to input before output

As mentioned in 2.4.3 a common false attack in ProVerif is the case where a term is used as input before the term has been output. A trivial example of this is the process $\nu k.pub\langle x\rangle.\overline{pub}\langle k\rangle.\mathit{if}\ x{=}k\ \mathit{then}\ P\ \mathit{else}\ Q$. By application of the operational semantics we note that to reach P we require to input k before it is known by the adversary, which is impossible. Due to the abstractions of ProVerif, it is possible for the adversary to input the key k at $pub\langle x\rangle$ and reach P. This may cause a false attack on the property we want to verify, for example if the property breaks when P is executed.

False attacks like this can appear on both private and public channels as both are translated into Horn clauses. In the next section we describe a method to avoid false attacks caused by replay of messages on private channels. That method also avoids false attacks caused by input before output in private channels.

The method for public channels is more complicated because of the presence of an adversary. The process that is to be encoded also has some requirements that may be hard to verify. The reason we add the requirements are to ensure that the adversary is not weakened by the use of our encoding.

We describe the intuition behind the modification, then the actual method and the requirements. Assume that k in our trivial example of the false attack is split into two parts k_1, k_2 where k_1 uniquely identifies k and k_2 is random data. The random data in k_2 can then identify at which point k was disclosed and used to avoid false attacks.

To split k into two parts, we introduce a function $k(k_1, k_2)$ that binds the first and second part into one term. When introducing the term $k(k_1, k_2)$ we need to extend our equational theory to allow extraction of k_1 and k_2 from the term. Therefore we add two equations to our equational theory, $first(k(x, y)) = x$ and

$second(k(x, y)) = y$, we assume that these are private equations, thus not usable by the adversary. Thus, the adversary cannot break a pair of $k(k_1, k_2)$ into smaller pieces.

Similarly to the example in the beginning of this section, it is impossible to reach P in the process $\nu k_1.\nu k_2.pub\langle x\rangle.\overline{pub}\langle k(k_1, k_2)\rangle$. *if* $first(x) = k_1$ *then (if* $second(x) = k_2$ *then* 0 *else* P*) else* Q.

For the first if statement to be true, the adversary would need to know the pair $k(k_1, k_2)$ before it is possible for her to know it. The difference from the example in the beginning of this section is that this process does not allow the abstractions of ProVerif to reach P. As the adversary cannot split a term $k(k_1, k_2)$, the only pair $k(k_1, k_2)$ that the adversary can input in the abstraction is the pair where both k_1 and k_2 match, thus leading to the 0 part of the process.

If we at some point create k_1 and for each usage change k_2, we can ignore all inputs where k_2 equals a version created by ourselves and not output yet. Any other combination of k_1, k_2 must be allowed during input.

As the k_2 in $k(k_1, k_2)$ may be changed, the adversary cannot identify that two terms are supposed to be equal in the original process. If this modification is to be applied one must argue that the adversary does not need to know if two terms derived from k are equal to perform an attack.

We assume that this can be achieved by duplicating all equations that may handle terms that may contain k, and modifying the duplicated equation to ignore k_2 of k. The motivation of this assumption is that the adversary in the Dolev-Yao model does not need to know that two terms are equal to be able to abuse the fact that they are.

The use of this method to avoid false attacks can be eased if the specific input and output operations that cause the false attack is identified. In this case only that output needs to generate a new k_2 and only the matching input needs to add a restriction on the k_1 and k_2 combination. All other inputs and outputs can use $k(k_1, k_2)$ as before with the modification that k_2 should be ignored when comparing two different $k(..)$ terms.

We have identified that this modification quickly becomes cumbersome to use if the term is used in a large portion of the protocol.

4.1.2 Avoiding false attacks in private channels

In chapter 2 we explained that private channels are completely protected from the adversary. The operational semantics in the same chapter show that output operations are blocking until a matching input operation can be executed. Both of these properties are also true in ProVerif.

When ProVerif converts the protocol into Horn clauses, the latter distinction disappears. In the created Horn clauses output on a private channel is not blocking. As a clause may be applied multiple times, a single output may also be used for multiple inputs.

A small example causing this false attack can be seen in figure 4.1. A single

$$\nu c.\nu s.\overline{c}\langle s\rangle.event(done(s))$$

Figure 4.1. Example of false attack caused by non blocking output.

$$c\langle x\rangle.P \Rightarrow \nu n.\overline{c}\langle n\rangle.c\langle{=}n, M\rangle.P$$
$$\overline{c}\langle M\rangle.P \Rightarrow c\langle n\rangle.\overline{c}\langle n, M\rangle.P$$

Figure 4.2. Avoiding false attacks on private channels.

$$\nu c.\nu s.((\overline{c}\langle s\rangle)|(c\langle{=}s\rangle.c\langle{=}s\rangle.\overline{pub}\langle s\rangle))$$

Figure 4.3. Trivial process that sends a secret over a private channel, if it is received twice it is output on the public channel to the attacker. As the out operation may be repeated in Horn clauses, this results in a false attack in ProVerif.

output on a private channel is followed by an event. This process has no inputs on the private channel and can never reach the event. ProVerif finds a trace where the event is reached, which is then identified as a false attack.

In figure 4.2 the modification for private channels is shown. For private channels we convert each input and output by adding nonces.

Input and output operations are extended with a nonce hand-off to avoid false attacks caused by replay of messages. Messages sent in previous communications will have the wrong nonce and cannot be used.

This translation does not modify the properties of the private channel, because it already had the property that a message cannot be sent without somebody waiting to receive. The nonces only enforce this by making the sending process choose who should receive the message by selecting their nonce.

During our work we noted that this modification may affect termination of ProVerif. In some tests addition of nonces caused non-termination, while in other cases addition of nonces helped with termination. Due to time limitations we have not tried to identify what caused this difference in termination.

4.1.3 Avoiding false attacks caused by repeated operations

We have previously mentioned specific ways to avoid some of the false attacks caused by the abstraction to Horn clauses. In this section we try to give a more general view of the problem and how to avoid the false attacks.

$$\nu c.\nu s.((\nu n_1.\overline{c}\langle n_1, s\rangle)|(c\langle n_2, {=}s\rangle.c\langle n_3, {=}s\rangle.\mathit{if}\ n_2 = n_3\ \mathit{then}\ 0\ \mathit{else}\ \overline{pub}\langle s\rangle))$$

Figure 4.4. Example of input and output that use IDs to avoid false attacks due to repetition of events. The original process can be seen in figure 4.3

This method is applicable when verifying a property that depends on two terms which are related in some way. Freshness is one example as two terms are required to break freshness. To avoid false attacks during a freshness check, we check if the terms were created due to a false attack.

To get an idea of the problem, we begin with an example. Figure 4.3 specifies a protocol where one party outputs a secret s on a private channel and another party tries to input the secret twice on the private channel. If it succeeds it outputs the secret on the public channel.

ProVerif will find a false attack when querying for the secrecy of s as the Horn clause representing the output can be applied multiple times. The method of avoiding false attacks on private channels does not work here as the false attack still exists if the encoding in section 4.1.2 has been applied.

One method to avoid this false attack is shown in figure 4.4. A new nonce is added to each output, and before disclosing the secret to the adversary we check that the two nonces we received were from different outputs. Each message has a new nonce, if the same nonce is received twice this is caused by the abstraction to Horn clauses and is therefore a false attack.

The core idea of this modification is to add information to the Horn clauses and add a required Horn clause which can only be applied if the added information does not represent a false attack.

Assume that we have a false attack where operation $A(M)$ is executed twice even though it is not possible in the process. We can generate an unique ID νi for the operation, and pass that ID forward by changing the operation to $A(i, M)$. To avoid the false attacks we specify requirements on the IDs that have been used through the protocol. Using this method it is possible to enforce requirements on operations and their order.

To relate this to figure 4.4, the name n_1 is used to uniquely identify a specific output. The inputs are modified to input IDs n_2 and n_3. To ensure that the duplicate input was not caused by the Horn clause abstraction a if statement is added that checks that the received IDs n_2 and n_3 are different.

Most Horn clauses are generated from input and output of messages, and contain information of which channel and what message was sent. We use messaging on private channels as an example to avoid false attacks; to use public channels is more complex as it requires one to account for the adversary.

To avoid false attacks where multiple outputs and inputs interact in ways not allowed by the protocol, a list of operation IDs can be passed through the protocol. The list is then checked for false attacks by using predicates. An example of this can be seen in chapter 5 where we verify freshness in MiniDC.

As this method requires us to add the list of process IDs to each message, we must ensure that the modification does not affect the properties that are to be verified.

Sometimes it may be easy to argue that adding the list to messages does not affect the protocol. For example if the protocol is tagged, then it is easy to identify where each output term can be input. A tagged protocol is a protocol where each

$$\begin{array}{c} \nu s.\nu I.\nu A.\nu B. \\ ([s \rightarrow\ I]| \\ (lock\ s.read\ s\ as\ x.if\ x = I\ then\ s := A.event(A).unlock\ s\ else\ 0)| \\ (lock\ s.read\ s\ as\ x.if\ x = I\ then\ s := B.event(B).unlock\ s\ else\ 0)) \end{array}$$

Figure 4.5. Example of process with false attack. A query for $event(A) \wedge event(B)$ would succeed in the Horn clause representation.

output has a unique term that specifies at which point in the process it can be input.

This method of avoiding false attacks may be hard to implement as it requires understanding of the generated Horn clauses. When implementing this method one has to prove that no real attacks are removed and also that the modified specification does not change the relevant properties of the original protocol.

4.2 Encoding state and freshness

In this section we apply the methods described above to the encodings described in chapter 3. We also add restrictions on the encodings to achieve termination in ProVerif.

4.2.1 Private state cells

The restriction we add to state cells is that the number of possible state transitions must be limited. That is, the number of times a state is locked and unlocked must be finite. For example there may not exist any infinite loops, and only a finite number of state values may exist. If our state cell consists of an increasing counter, the latter restriction forces us to have a maximum value of the counter and the former restriction does not allow us to decrease or reset the counter.

The reason we have this restriction is that an infinite number of state transitions usually cause non-termination in ProVerif.

False attacks appear when using the previously described encoding for private state cells in ProVerif. The reason is that information is never lost during ProVerifs Horn clause resolution. When a clause describing the transaction from state X to state Y is applied, knowledge about state Y is added, but the knowledge about state X is not removed. Thus the state X and state Y exist at the same time even though this should not be possible.

An example of a false attack due to state can be seen in figure 4.5. $event(A)$ and $event(B)$ can both occur separately, but not in the same trace. During the conversion to Horn clauses ProVerif adds one rule describing that from state I we can reach state A and another, describing that we can reach state B from state I. The problem is that both these rules can be used as even though we have modified the state the original state still exists in the Horn clauses. Thus, ProVerif can first

apply the former rule, and then the latter rule and think that both $event(A)$ and $event(B)$ are reachable in the same trace.

False attacks due to this can be hard to avoid and generally needs a modification of the process where a list of state IDs are passed along with the state. A general method of this was discussed in section 4.1.3.

Another cause of false attacks on private state cells is the use of private channels. These can be avoided by using a nonce hand-off for messaging, as described in 4.1.2.

In the next chapter we describe how this method can be used in practice. In appendix A we show a ProVerif model of MiniDC that uses this method.

4.2.2 Encoding freshness

The encodings in applied π-calculus for freshness from the previous chapter works in ProVerif. The encoding in itself does not cause any false attacks and trivial cases of freshness are often possible to prove.

An example of a trivial case is when the term we check freshness for contains at least one term which was freshly generated in the checking process.

In non-trivial cases we, in practice always, get false attacks due to the abstractions made by ProVerif. If freshness of a term depends on a Horn clause only being used once, then this causes a false attack. This can be avoided by the list method described in 4.1.3. We repeat that this method can be complicated to apply and it may be hard to verify that the modification is sound.

We remind the reader that even though we get false attacks, the possibility of finding a real attack still remains. This means that freshness in a protocol with an actual attack can sometimes be proven not to hold by using this encoding.

In the next chapter we show how to verify freshness in MiniDC.

Chapter 5

Verification of MiniDC

To verify freshness of the key received by Bob in MiniDC we have to perform some modifications. We use one of the ProVerif specific features to verify the protocol. The feature we use is the possibility of creating predicates in ProVerif. This feature is used to introduce checks on the lists created to prevent false attacks as mentioned in 4.2.1. These predicates are only described informally in this section, while the complete modified ProVerif input can be found in appendix A.

5.1 Modifications to allow encoding

To apply the encodings and avoid false attacks we first modify our protocol to follow the requirements. As stated in section 4.2.1 we need to have a finite number of state transitions to achieve termination in ProVerif. Because of this, we limit MiniDC to ten increments of the counter n.

It should be stressed that by modifying the protocol this way we are in fact performing bounded verification because our initial version allowed infinite values for the counter. Although many protocols have a maximum value for counters, this maximum value is often higher than what this encoding in ProVerif allows us to verify without an unacceptably long execution time. This can be seen in the performance test in the end of this chapter.

5.2 Applying encodings

The first step we take is to apply both the encoding of state and the encoding for freshness to the model. The result can be seen in figure 5.1. We have previously explained that it is not enough to apply the encodings as the encoding for state causes false attacks. MiniDC uses a private channel to deliver the key to Bob, as private channels cause false attacks on freshness we must modify the protocol to avoid them.

The first method to avoid false attacks that we apply is to use nonces on private channels to avoid replay attacks. This method is explained in section 4.1.2 and is

Operation	**Description**
$\nu c_f.$	Create freshness channel c_f
$\nu s_f.$	Create freshness secret s_f
$\nu c.$	Create private channel c
$\nu zero.$	Create constant zero
$\nu s.$	Create secret s
$\nu c_s.$	Create state cell c_s
$\overline{c_s}\langle zero\rangle \mid$	Initialize state
!((	Replicated Alice process
$c_s\langle val\rangle.$	Lock+Read state into val
$\overline{c}\langle keygen(val, s)\rangle.$	Generate key and sent do Bob
$if\ val = inc10(zero)\ then\ 0\ else$	Short for inc(inc(ten times
$\overline{c_s}\langle inc(val)\rangle)$	Write+Unlock state with inc(val)
|(	Replicated Bob process
$c\langle key\rangle.$	Receive key
νn_f	Create nonce used for freshness
$(\overline{c_f}\langle n_f, key\rangle \mid$	Output key to freshness channel
$c_f\langle n'_f, {=}key\rangle.$	Receive key from freshness channel
$if\ n' = n\ then\ 0\ else$	Stop if message was produced by ourselves
$\overline{pub}\langle s_f\rangle)$	Output freshness secret
))	

Figure 5.1. MiniDC where encodings are applied

applied by adding a nonce hand-off to communication on private channels.

In figure 5.2 this method has been applied. We have not modified the communication on the private channel for freshness. This is because the encoding for freshness does not suffer from false attacks, thus we do not need to avoid them for this channel.

The modified version still suffers from false attacks. These are caused by the encoding of state. To avoid these we create a list of nonces where one nonce is added for each state transition. The list is passed forward with the messaging of the state. This modification causes all Horn clauses created from the state to also contain the list of nonces, which we can use to avoid false attacks. The lists are checked for false attacks when considering if two terms break freshness.

A working version can be seen in figure 5.3. In this version we added a predicate $disOrSub(listA, listB)$ which operates on lists. This predicate returns true if two lists have no elements in common or if one of the lists is a true prefix for the other. We only have one state and use locks to modify it in sequence. This results in the fact that if two lists are created during the passing of this state, one must be a true prefix of the other.

If for example $listA = [1, 2, 3]$ and $listB = [1, 2, 4]$ we know this is a false attack as we cannot have two different lists of the same length due to the way we use our single state cell. If the lists are of different lengths, the shorter list must be a prefix of the longer one as otherwise there must have been a false attack at the point where the lists were of equal length. We require one of the lists to be a true prefix of the other. This avoids cases where the lists are of equal length and have the same elements. This is a false attack as we only generate a single key for each state value.

After this addition, we note that the nonce list also avoids the false attacks caused by a process instance communicating with itself. Thus, one may be tempted

Operation	Description
$\nu c_f.$	Create freshness channel c_f
$\nu s_f.$	Create freshness secret s_f
$\nu c.$	Create private channel c
$\nu zero.$	Create constant zero
$\nu s.$	Create secret s
$\nu c_s.$	Create state cell c_s
$(c_s\langle n_a\rangle.\overline{c_s}\langle n_a, zero\rangle)\vert$	Initialize state
!((	Replicated Alice process
$\quad \nu n_b.\overline{c_s}\langle n_b\rangle$	Force ordering on private channel
$\quad c_s(=n_b, val).$	Lock+Read state into val
$\quad c\langle n_b\rangle$	Force ordering on private channel
$\quad \overline{c}\langle n_b, keygen(val, s)\rangle.$	Generate key and sent do Bob
$\quad$ *if* $val = inc10(zero)$ *then* 0 *else*	Short for inc(inc(ten times
$\quad c_s\langle n_c\rangle.$	Force ordering on private channel
$\quad \overline{c_s}\langle n_c, inc(val)\rangle)$	Write+Unlock state with inc(val)
|(	Replicated Bob process
$\quad \nu n_d.\overline{c}\langle n_d\rangle$	Force ordering on private channel
$\quad c(=n_d, key).$	Receive key
$\quad \nu n_f$	Create nonce used for freshness
$\quad (\overline{c_f}\langle n_f, key\rangle\vert$	Output key to freshness channel
$\quad c_f\langle n'_f, =key\rangle.$	Receive key from freshness channel
$\quad$ *if* $n' = n$ *then* 0 *else*	Stop if message was produced by ourselves
$\quad \overline{pub}\langle s_f\rangle)$	Output freshness secret
))	

Figure 5.2. MiniDC where encodings are applied and nonces added to communication on private channels.

to remove the nonces used during communication of state and only rely on the list. A modification like that results in non-termination of the clause solver in ProVerif.

5.3 Verification results

For a maximum value 12 of the counter in the running example, we get a result that no attacks exist. In practice this means that 13 keys can be sent from Alice to Bob without breaking freshness. This is an expected result. To check for trivial errors in our model we try to verify it for the case where Alice always generated the key from the value 'zero'. This test gives a 'cannot be proven' result, which indicates that ProVerif managed to find the attack in the Horn clause representation but failed to verify that the attack existed in the original description of the protocol.

5.4 Performance

We measured how much time it takes for ProVerif to verify variations of MiniDC on an Intel Core i7-4800MQ 2.70GHz under Linux 3.13 (Ubuntu) with 16Gb ram.

As expected, when using the list method to avoid false attacks the running time is exponential. As seen in figure 5.4, verification of MiniDC with a maximum counter of 12 takes two hours. We expect verification of a maximum value of 13 to take 25h. After 13h, we aborted the verification attempt for a maximum value of 13.

Operation	**Description**
$\nu l_0.$	Create empty list constant
$\nu c_f.$	Create freshness channel c_f
$\nu s_f.$	Create freshness secret s_f
$\nu c.$	Create private channel c
$\nu zero.$	Create constant zero
$\nu s.$	Create secret s
$\nu c_s.$	Create state cell c_s
$(\nu l_a.c_s\langle n_a\rangle.$	
$\quad \overline{c_s}\langle n_a, zero, add\_list(l_a, l_0)\rangle$	Initialize state
$)\mid$	
$!($(	Replicated Alice process
$\quad \nu n_b.\overline{c_s}\langle n_b\rangle$	Force ordering on private channel
$\quad c_s\langle{=}n_b, val, list\rangle.$	Lock+Read state into val
$\quad c\langle n_b\rangle$	Force ordering on private channel
$\quad \overline{c}\langle n_b, keygen(val, s), list\rangle.$	Generate key and sent do Bob
$\quad$ *if* $val = inc10(zero)$ *then* 0 *else*	Short for inc(inc(ten times
$\quad c_s\langle n_c\rangle.$	Force ordering on private channel
$\quad \nu l_b.$	Create new list element
$\quad \overline{c_s}\langle n_c, inc(val), add\_list(l_b, list)\rangle)$	Write+Unlock state with inc(val)
$\mid($	Replicated Bob process
$\quad \nu n_d.\overline{c}\langle n_d\rangle$	Force ordering on private channel
$\quad c\langle{=}n_d, key, list\rangle.$	Receive key and nonce list
$\quad (\overline{c_f}\langle key, list\rangle\mid$	Output key to freshness channel
$\quad c_f\langle{=}key, list'\rangle.$	Receive key from freshness channel
$\quad$ *if* $disOrSub(list, list')$ *then*	Check if this is a valid attack
$\quad \overline{pub}\langle s_f\rangle$	Output freshness secret
$\quad$ *else* 0)	If false attack, do nothing
$))$	

Figure 5.3. MiniDC where encodings are applied and false attacks caused by the encoding of state are avoided.

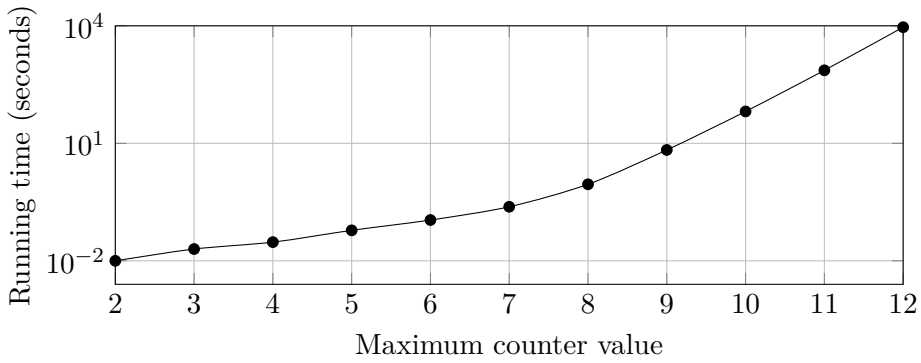

Figure 5.4. Running time when verifying freshness.

If the key sent from Alice also contains a new name the verification time grows polynomially. Verification of this version with a maximum counter of 38 takes 57.91 seconds. This can be seen in the figure 5.5.

If the new name is generated right before the output of the key we can remove the list and not have any false attacks. This version can still not be verified without an upper limit on the counter as it leads to non-termination. Verification of a counter with max value 99 takes 2.5 seconds. The plot 5.6 also suggests polynomial growth, although with a slower rate. Note that as the value of the state is not relevant for

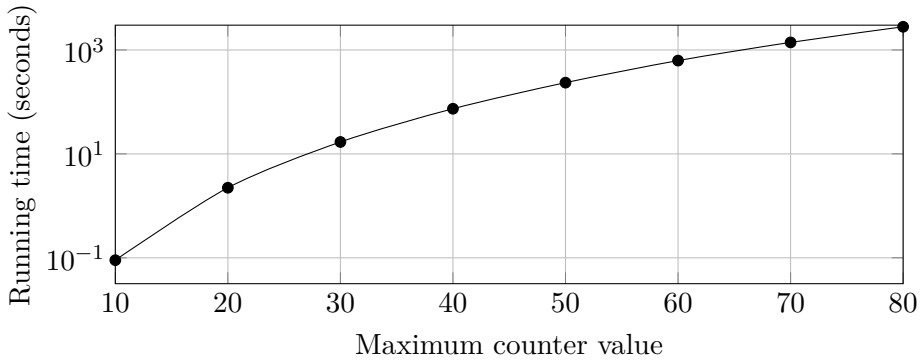

Figure 5.5. Running time when verifying freshness where fresh name is added to key before output from Alice.

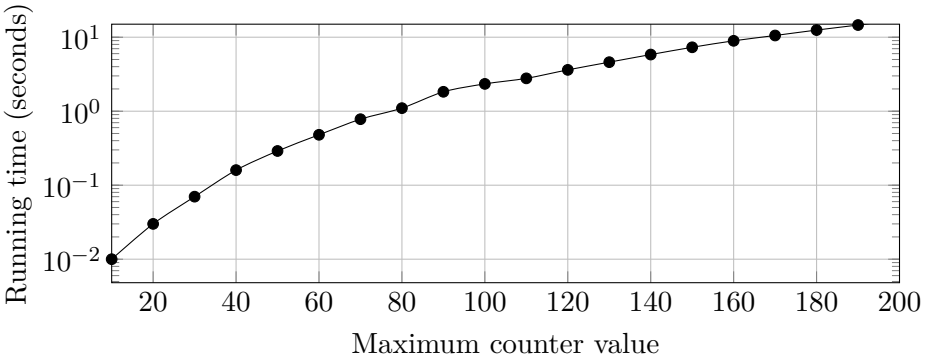

Figure 5.6. Running time when verifying freshness without list.

freshness in this version, state can be removed from the protocol when verifying freshness. The generated key can not become more random by removing sources of randomness, thus if freshness fails with state it will also fail without state.

Chapter 6

Conclusion

In this thesis we have investigated how to encode state in the applied π-calculus and how this encoding can be used in ProVerif. As ProVerif makes abstractions to achieve termination it is no surprise that if we work around issues caused by the abstraction we may get non-termination.

We have tried the encodings on a trivial protocol where state was used as a counter. As our experiments show, the time it takes to verify this trivial protocol is high even when limited to one single iteration with a maximum value of 12. We conclude that more complex protocols with state are infeasible to verify with this method.

The encoding for state causes false attacks when verifying freshness. This is because the same state may be used multiple times. To avoid false attacks we have shown how a restriction on a sequence of states can be enforced. This restriction requires us to have a clear way to pass additional information with the state value and a point where we can verify a requirement on the combination of state values that cause the false attack.

Verifying our trivial protocol using this method was cumbersome. Employing all methods to avoid false attacks is not simple even in cases when only private channels are used. When public channels are used, one must also ensure that an adversary is not weakened by the modifications.

6.1 Results

Our main result is that it is possible to encode state and freshness in the applied π-calculus and that it is possible to use these encodings in ProVerif while avoiding false attacks.

We have identified three methods to avoid false attacks when verifying freshness, all based on the use of nonces. False attacks introduced by repeated messages on private channels can be avoided using a nonce hand-off. This method is similar to how nonces are used in encryption to avoid replay attacks. False attacks introduced by input before output of a secret can be avoided by adding information to the

secret. This information can be used to determine at which point in the protocol the secret was disclosed. Finally, to avoid false attacks caused by the encoding of state, a nonce list method can be applied. This method requires that there exists a position in the protocol where the relation between two terms can identify a false attack.

When verifying MiniDC with ProVerif and these encodings, we are limited to bounded verification. Allowing 12 keys to be generated requires a running time of 2 hours on an Intel Core i7-4800MQ 2.70GHz under Linux 3.13 (Ubuntu) with 16Gb ram. It is apparent that even for this trivially correct protocol, automatic verification in ProVerif is complicated. This encoding of state is not applicable for verification of more complicated protocols.

6.2 Discussion

Among our three methods to avoid false attacks, the one for private channels is the least intrusive and easiest to implement. For this method there are no additional requirements and the method can be applied without additional work. Using the method can affect running time and termination of ProVerif in both directions, it may increase running time or decrease it.

To avoid false attacks due to input before output of a secret, the secret is bound with information which identifies when the secret was output. This binding weakens the adversary as she is no longer able to distinguish if two secrets are equal. To use the encoding the user must ensure that no attack is lost due to this.

The method of creating a nonce list to avoid false attacks is the most complicated. This method is both hard to apply and it is hard to verify that the application is correct. Lists of nonces are created during execution of the protocol. These lists can be viewed as partial traces, and can be compared to identify traces which are caused by the abstractions of ProVerif.

As this method requires the lists to be passed through the protocol, messaging must be altered to include them. The point where the lists are compared must be carefully chosen to avoid false attacks. The comparison must be able to stop false attacks while not stopping true attacks. The method has been shown to make verification in ProVerif slow: our running example limited to a list of length 12 takes 2 hours to verify.

From our performance tests in the case study we note that the running time can be low even when we use the nonce list method. In our case this was because the nonce list was not relevant for the verification.

6.3 Future work

Some ideas expressed here may be trivial or impossible as we have done no research for them yet.

As tools use different verification methods, they are good for different types of protocols and properties. A tool employing multiple methods could for example try to verify one property for the same protocol using multiple methods and take the result of the method that succeeds. This result could be used to guide the tool when checking other properties.

Similarly, more formal work could be done on identifying how the different verification methods and algorithms compare depending on protocol and properties.

It would be beneficial to have a common input language that can be used in more than one tool. The applied π-calculus seems to be a good candidate. Although it may be far from the implementation language it seems like a good compromise between expressiveness and readability.

As verification of protocols is hard, composition of protocols could be a solution. Small protocols often make verification simple. One problem with this approach is that it is not applicable to existing protocols. Another issue with this approach may not be applicable for protocols designed for limited hardware.

ProVerif sometimes has trouble finding actual attacks: if an attack exists it is common to get 'cannot be proven' as result. As an experiment we added some randomization to ProVerif's Horn clause selection heuristic and managed to find true attacks for protocols that had a 'cannot be proven' result previously. We think that there could be done much to improve the actual attack finding part of ProVerif.

If ProVerif is extended to allow additional information to be linked to a term in a way that allows one to modify and add information, this could be used to avoid specific false attacks. The linked information should not be accessible to the adversary and not impact use of the term during equality checks.

6.4 Ethics and sustainability

The method used to perform bounded verification of MiniDC has exponential running time. To motivate usage of this method one needs to argue that the value of the result is higher than the value of the resources used for verification. Security protocols can reduce the need of long distance travel. If a security protocol significantly reduces long distance travel, it may be justified to verify it using this method.

Our results do not have any significant ethical implications. We do not prove that it is impossible to verify security protocols with state nor do we introduce an effective way of verifying such.

Bibliography

[1] R. M. Needham and M. D. Schroeder. "Using Encryption for Authentication in Large Networks of Computers". In: *Communications of the ACM* 21.12 (1978).

[2] G. Lowe. "Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR". English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by T. Margaria and B. Steffen. Vol. 1055. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996. ISBN: 978-3-540-61042-7.

[3] A. Armando et al. "Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-based Single Sign-on for Google Apps". In: *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering*. FMSE. Alexandria, Virginia, USA: ACM, 2008. ISBN: 978-1-60558-288-7.

[4] D. Dolev and A. C. Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983).

[5] N. A. Durgin et al. "Undecidability of Bounded Security Protocols". In: *Proceedings of the Workshop on Formal Methods and Security Protocols (FMSP)*. 1999.

[6] R. M. Amadio and W. Charatonik. "On Name Generation and Set-Based Analysis in the Dolev-Yao Model". In: *CONCUR 2002 - Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002, Proceedings*. 2002.

[7] N. A. Durgin et al. "Multiset Rewriting and the Complexity of Bounded Security Protocols". In: *Journal of Computer Security* 12 (2002).

[8] M. Abadi and C. Fournet. "Mobile Values, New Names, and Secure Communication". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '01. London, United Kingdom: ACM, 2001. ISBN: 1-58113-336-7.

[9] S. Meier. "Advancing automated security protocol verification". PhD thesis. Zürich, 2013.

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

[11] *3GPP System Architecture Evolution; TS 33.401*. V12.14.0. 3GPP. 2015. URL: http://www.3gpp.org/ftp/Specs/archive/33_series/33.401/33401-ce0.zip.

[12] N. B. Henda, K. Norrman, and K. Pfeffer. "Formal Verification of the Security for Dual Connectivity in LTE". In: *Proceedings of the 3rd FME Workshop on Formal Methods in Software Engineering*. FormaliSE 2015. 2015.

[13] G. Lowe. "A Hierarchy of Authentication Specifications". In: *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*. CSFW. Washington, DC, USA: IEEE Computer Society, 1997. ISBN: 0-8186-7990-5.

[14] C. Boyd and A. Mathuria. *Protocols for Authentication and Key Establishment*. Information Security and Cryptography. Springer, 2003. ISBN: 978-3-642-07716-6.

[15] U. M. Maurer and P. E. Schmid. "A Calculus for Security Bootstrapping in Distributed Systems". In: *Journal of Computer Security* 4.1 (1996).

[16] I. Cervesato et al. "A meta-notation for protocol analysis". In: *Computer Security Foundations Workshop, 1999. Proceedings of the 12th*. IEEE, 1999.

[17] D. Basin, C. Cremers, and C. Meadows. "Model Checking Security Protocols". English. In: *Handbook of Model Checking*. Springer, To appear. Chap. 24.

[18] A. Armando, R. Carbone, and L. Compagna. "SATMC: a SAT-based Model Checker for Security-critical Systems". In: *TACAS'14: Proceedings of the 20th international Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Grenoble, France: Springer, 2014. ISBN: 978-3-642-54861-1.

[19] M. Arapinis et al. "StatVerif: Verification of stateful processes". In: *Journal of Computer Security* 22.5 (2014).

[20] B. Schmidt. "Formal analysis of key exchange protocols and physical protocols". PhD thesis. Zürich, 2012.

[21] S. Kremer and R. Kunnemann. "Automated Analysis of Security Protocols with Global State". In: *Security and Privacy (SP), 2014 IEEE Symposium on*. May 2014.

[22] M. Abadi and A. D. Gordon. "A Calculus for Cryptographic Protocols: The spi Calculus". In: *Information and Computation* 148.1 (1999).

[23] R. Milner, J. Parrow, and D. Walker. "A Calculus of Mobile Processes, I". In: *Information and Computation* 100.1 (1992).

[24] R. Milner, J. Parrow, and D. Walker. "A Calculus of Mobile Processes, II". In: *Information and Computation* 100.1 (1992).

[25] M. Arapinis et al. "Stateful Applied Pi Calculus". English. In: *Principles of Security and Trust*. Ed. by M. Abadi and S. Kremer. Vol. 8414. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014. ISBN: 978-3-642-54791-1.

[26] T. Chothia, B. Smyth, and C. Staite. “Automatically Checking Commitment Protocols in ProVerif without False Attacks”. English. In: *Principles of Security and Trust*. Ed. by R. Focardi and A. Myers. Vol. 9036. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015. ISBN: 978-3-662-46665-0.

[27] B. Nguyen and C. Sprenger. “Abstractions for Security Protocol Verification”. English. In: *Principles of Security and Trust*. Ed. by R. Focardi and A. Myers. Vol. 9036. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015. ISBN: 978-3-662-46665-0.

Appendix A

ProVerif code for verified MiniDC version

```
set verboseClauses = short.
type nonce.
type state.
type noncelist.
type counter.

free pub:channel.
free f_ch:channel [private].
free c:channel [private].
free s_cell:channel [private].

const zero:counter.
const f_secret:bitstring [private].
const secret:bitstring[private].
const s_emptyset:noncelist [data].


query attacker(f_secret).


fun keygen(counter, bitstring):bitstring.
fun inc(counter) : counter.
        reduc forall x:counter; dec(inc(x)) = x.

fun statedata(nonce, counter, noncelist):state [data].
fun add_list(nonce, noncelist): noncelist [data].

pred s_mem(nonce, noncelist) [memberOptim].
pred s_notmem(nonce, noncelist).
pred s_disjunct(noncelist, noncelist).
pred s_issubset(noncelist, noncelist).
pred s_disOrsub(noncelist, noncelist).
```

```
clauses
(* s_mem - just as mem() for sets *)
    forall x: nonce, y: noncelist; s_mem(x, add_list(x, y));
            forall x: nonce, y: noncelist, z: nonce; s_mem(x, y) → s_mem(x, add_list(z, y));
(* s_notmem - not(mem()) for sets *)
    forall x: nonce; s_notmem(x, s_emptyset);
            forall x: nonce, y: noncelist, z: nonce; s_notmem(x, y) ∧ x ≠ z →
            s_notmem(x, add_list(z, y));
(* s_disjunct - two lists have no elements in common *)
    forall xt:noncelist; s_disjunct(xt, s_emptyset);
    forall xt:noncelist, yt:noncelist, ye:nonce;
        s_notmem(ye, xt) ∧ s_disjunct(xt, yt) → s_disjunct(xt, add_list(ye, yt));

(* s_issubset - (Actually 'is_true_subset' as it returns false for the same set) ,
issubset(A,B) check if B is true subset of A *)
    forall x:nonce, xt:noncelist; s_mem(x, add_list(x, xt)) → s_issubset(add_list(x, xt), xt);
    forall x:nonce, xt:noncelist, yt:noncelist; s_issubset(yt, xt) ∧ s_mem(x, add_list(x, yt)) →
    s_issubset(add_list(x, yt), xt);

(* s_disOrsub - Returns true if the lists are completely disjunct or if one is a subset
of the other. *)
    forall xt:noncelist, yt:noncelist, ye:nonce;
        s_disjunct(xt, yt) → s_disOrsub(xt, yt);
    forall xt:noncelist, yt:noncelist, ye:nonce;
        s_disjunct(yt, xt) → s_disOrsub(xt, yt);
    forall xt:noncelist, yt:noncelist, ye:nonce;
        s_issubset(xt, yt) → s_disOrsub(xt, yt);
    forall xt:noncelist, yt:noncelist, ye:nonce;
        s_issubset(yt, xt) → s_disOrsub(xt, yt).



let Alice() =
            new n:nonce;
    out(s_cell, n);
    in(s_cell, statedata(= n, val, list));

    in(c, n2:nonce);
            out(c, (n2, keygen(val, secret), list));            (* output the generated key *)

    if val ≠ inc(inc(inc(inc(inc(inc(inc(inc(inc(zero)))))))))) then
        in(s_cell, n3:nonce);
        out(s_cell, statedata(n3, inc(val), add_list(n3, list))).

let Bob() =
```

```
    new n:nonce;

    out(c, n);
          in(c, (= n, key:bitstring, list:noncelist));

    out(f_ch, (key, list)) | (
    in(f_ch, (= key, list2:noncelist));

    if s_disOrsub(list, list2) then
       out(pub, f_secret)
    ).

let init_state(init:counter) =
    new myn:nonce;
    in(s_cell, n:nonce);
    out(s_cell, statedata(n, init, add_list(myn, s_emptyset))).

process init_state(zero) |!(Bob() | Alice())
```