A Formalization of Strand Spaces in Coq

Project Number: DJD-AAOA

A Major Qualifying Project

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Hai Nguyen

Date: May 2015

APPROVED:

Professor Daniel Dougherty, MQP Advisor

Abstract

In this paper we formally prove the correctness of two theorems about cryptographic protocol analysis by using the Coq proof assistant. The theorems are known as the Authentication Tests in the strand space formalism. With such tests, we can determine whether certain values remain secret so we can check whether certain security properties are achieved by a protocol. Coq is a formal proof management system. It provides a formal language to express mathematical assertions, mechanically checks proofs of these assertions. Coq works within the theory of the calculus of inductive constructions, which is a variation on the calculus of constructions. We first formalize strand spaces by giving definitions in Coq of the basic notions. Then we express the two authentication tests and give constructive proofs for them.

Acknowledgements

I would like to express my gratitude to my advisor, Professor Daniel J. Dougherty, for his outstanding support through the project.

Thanks also to Professor Joshua Guttman for his quick feedback whenever I have any questions about strand spaces, and authentication tests.

Thanks also to lots of friends, the fact that a week has seven days instead of only five as I had always thought, and the fact that I own a key to the building so I can work at four in the morning whenever I feel like it. That is, all the time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives a short introduction about project motivations, cryptographic protocols, and proof assistant.

1.1 Objectives and Project Motivations

Cryptographic protocols are intended to let principals communicate securely over a communication protocol which are designed to provide various kinds of security assurances. An important security goal of cryptographic protocol is authentication, the act of confirming the truth of an attribute of a datum or entity like verifying freshness of a nonce. Many research papers about authentication have been published. One of them is Authentication Tests and the Structure of Bundles by Joshua Guttman and Javier Thayer [6]. The main idea of authentication tests is that if a principal in a cryptographic protocol creates and transmits a message containing a new value v, and later receives v back in a different cryptographic context then it can be concluded that some principal processing the relevant key has received and transformed the message in which was emitted. The authentication tests themselves are easy to apply but the proof justifying them are more complicated [6]. Though authentication tests are proved in the paper, they have not been formally verified. As we know that once lemma or a theorem has been proved in some proof assistant language like Coq, we will have a very strong assurance that it is true - much more than what we usually have when doing a pen-and-paper proof. In addition, we found that there are few papers and projects using Coq to verify security goal of cryptographic protocols and to particularly formalize strand spaces, which is a well-known approach to cryptographic protocols.

In this project we prove authentication tests under strand space formalism approach using the Coq proof assistant. First, we formalize strand spaces and all basic concepts needed for proving authentications tests like components,transformation paths, penetrable keys. Then we provide detailed formal proofs of all relevant lemmas, theorems, and finally authentication tests.

The purpose of the project is to help researchers in security area have more confidence in using the result of authentication tests since they are formally verified. Our implementation is modular so that researchers can easily extract certain modules for their purpose. For example, the formalization can be used as a frame work for later research using strand space approach.

1.2 Reasoning about Cryptographic Protocols

Cryptographic protocols are programs that aim at securing communications on insecure networks, such as Internet, by relying on cryptographic primitives. Even when cryptographic protocols have developed carefully by experts and also reviewed thoughtfully by other experts, the design of cryptographic protocols may contain some bugs possibly causing them unusable [9]. For instance, in the Needham-Schroeder public-key protocol, a flaw (using man in the middle attack method) was found by Lowe 17 years after its publication. Although much progress has been made, current cryptographic protocols may still have some flaws. Moreover, security errors cannot be detected by functional software testing because they appear only in the presence of a malicious adversary. Automatic tools can therefore be very helpful in detecting and also verifying the correctness of security protocols. A lot of tools for verifying and analyzing cryptographic protocols have been developed like ProVerfi, SATMC, PVS, and CPSA. Hence, security protocol verification has been a very active research area since 1990s.

There are several techniques for proving protocol correctness. Two common approaches in this area are symbolic and computational models. Symbolic model approach relies on specifications while computational model approach relies on implementations. A well-known approach of the former one is strand space model, developed by Joshua D. Guttman, Javier Thayer Fabrega, and Jonathan C. Herzog [5]. This approach has several advantages as following.

- It gives a clear semantics to the assumption that certain data items, such as nonces and session keys, are fresh, and never arise in more than one protocol run [5].
- It provides an explicit model of the possible behaviors of a system penetrator; this allows to develop general theorems that bound the abilities of the penetrator, independent of the protocol under study [5].
- It allows various notions of correctness, involving both secrecy and authentication, to be stated and proved [5].

- The approach leads to detailed insight into the reasons why the protocol is correct, and the assumptions required. Proofs are simple and informative: they are easily developed by hand, and they help to identify more exact conditions under which we can rely on the protocol [5].

We will describe in details strand spaces in the next chapter.

1.3 Proof Assistants

Proof assistants (interactive theorem provers) are computer systems that allow a user to do mathematics on a computer, focusing on the aspects of proving and defining but not so much the computing. So a user can set up a mathematical theory, define properties and do logical reasoning with them. In many proof assistants one can also define functions and compute with them, but their main focus is on doing proofs. As opposed to proof assistants, there are also automated theorem provers. These are systems consisting of a set of well chosen decision procedures that allow formulas of a specific restricted format to be proved automatically. Automated theorem provers are powerful, but have limited expressivity, so there is no way to set-up a generic mathematical theory in such a system.

There are a lot of proof assistant systems like Isabelle, Coq, PVS, NuPRL. Out of them, Coq seems to be the most powerful system that supports a lot of features such as higher-order logic, dependent types, proof automation, proof by reflection, code generation. The Coq proof assistant is distinguished from the other. That is the main reason that we chose to use Coq instead of another proof assistant.

In addition, proving using Coq provides numerous advantages over paper-and-pencil proofs. First, Coq can mechanically check our proofs, hence it provides much greater confidence on our formalization and on the correctness of our theorems. Second, because all proofs in Coq are constructive, we can automatically extract certified implementations of all our theories. This provides runnable tools (for free!) and give us confidence in the tools as well. Finally, a mechanized representation is more valuable to others who can easily adapt our work to related projects and obtain high assurance in the results.

1.4 Related Work

This section shall describe some work that is related to my project.

1. The first one is "A formalization of the spi calculus in Coq" [4]. Spi calculus is an extension of Pi calculus. It is used to model and study cryptographic protocols. That project is similar to my project because it is also a formalization

of some mathematical structure in Coq. However, formalizing spi calculus and formalizing strand spaces have different styles. While spi calculus is a functional programming with concurrent processes [4], strand space model is about logic.

2. The second related work is Proving "Proving Security Protocols Correct" Correct: Formal Verification of Strand Spaces by Andrew Kent and J McCarthy. This work is very similar to my project since it is also to formalize strand spaces. In this work, they followed an other paper of Joshua Gutmman and Thayer Javier, which is just about strand spaces. So their goals are to formalize strand spaces, and to prove some properties of strand spaces; they did preliminary work but it remains unpublished. In my project, I have a different formalization of strand spaces in Coq and I did prove a lot of lemmas and theorems about strand spaces, and authentication tests.

3. CertiCrypt is a fully machine-checked framework built on top of the Coq proof assistant [7]. It is a tool that assists the construction and verification of cryptographic protocols. It supports common patterns for reasoning about cryptography, and has been used successfully to prove many security goals, for example, encryption, digital signature schemes, and zero-knowledge protocols [2]. CertiCrypt provides a rich set of verification techniques for probabilistic programs, including equational theories of observational equivalence, a probabilistic relational Hoare logic, certified program transformations, and techniques widely used in cryptographic proofs such as eager/lazy sampling and failure events [7]. CertiCrypt works in the "computational model" for protocol analysis, as opposed to the "symbolic model" that is the context of our work.

Chapter 2

Background

2.1 Strand Space Overview

In this section, we briefly summarize the ideas behind the strand space model. The Coq development in the next chapter will provide precise definitions.

A strand spaces is a set of strands; one may think of a strand space as containing all legitimate executions together with all the actions that a penetrator may apply to the messages contained in these executions.

A strand is a sequence of events that a single principal, either a legitimate principal or a penetrator, may engage in. The height of a strand is the number of nodes on that strand. Each strand is a sequence of message transmissions and receptions with specific values such as nonces and keys. Transmission of a term t is represented as $+t$ and reception of a term t is represented as $-t$. Each element of a strand is called a node. Given a strand s, (s, i) is the i^th node on s. We say that $n \Rightarrow n'$ if $n = (s, i)$ and $n' = (s, i + 1)$. Thus, the relation $\Rightarrow^+$ between two nodes is the transitive closure of the relation $\Rightarrow$. The relation $n \rightarrow n'$ represents the inter-strand communication; it means that $term(n) = +t$ and $term(n') = -t$; here $term(n)$ denotes the signed (unsigned) message at the node n.

Let A be the set of all possible messages that can be exchanged between principals in a protocol. We call elements of A terms. A is freely generated from two disjoint sets, set of texts T and set of cryptographic keys K, by concatenations $encr : K \times A \rightarrow A$ and encryptions $join : A \times A \rightarrow A$. Hence, A is closed under concatenation and encryption. The set K is equipped with an injective unary operator $inv : K \rightarrow K$ which maps each member of asymmetric key pair to the other and maps a symmetric key to itself.

A signed term is a pair of a sign $\sigma \in +, -$ and a term t, written either $< \sigma, t >$ or $+t$ or $-t$.

A term t_1 is a subterm of another term t_2, denoted as $t_1 \sqsubset t_2$, if we can get t_2 from t_1 by repeatedly concatenating with arbitrary terms and encrypting with arbitrary keys. For example, A, N_a are subterms of $|N_a A|_K$ but K is not.

Another important concept under strand space is origination. We say that a term t originates at a node n if n is a transmission node, $t \sqsubset term(n)$, and t is not a sub-term of any earlier node of n; hence, n is the first node in its strand includes t. A node is called uniquely originating if it is originated on only one node over all strands.

A bundle is a casually well-founded collection of nodes and two relations $\Rightarrow$ and $\rightarrow$. It represents the actual protocol interactions. In a bundle, when a a strand receives a message m, there is a unique node transmitting m from which the message was immediately received. In contrast, when a strand transmits a message m, many strands or none may immediately receive m. The height of a strand in a bundle is the number of nodes on the strand that are in the bundle.

The penetrator's powers are characterized by the set of compromised keys which are initially known to penetrator, and a set of penetrator strands that allow the penetrator to generate new messages. The set of compromised keys typically would contain all public keys, all private keys of penetrators, and all symmetric keys initially shared between the penetrator and principals playing by the protocol rules. The atomic actions available to penetrator are encoded in a set of penetrator strands. We partition penetrator strands according to the operations they exemplify. E-strands encrypt when given a key and a plain-text; D-strands decrypt when given a decryption key and matching cipher-text; C-strands concatenate terms; S-strands separate terms; M-strands emit known atomic text or guess; and K-strands emit keys from a set of known keys.

Important units for protocol correctness are components. A term t is a component of another term t' if $t \sqsubset t'$, t is nt a concatenated term, and for every $s \neq t$ such that $t \sqsubset s \sqsubset t'$, s is a concatenated term. Thus, a component is either atomic value or an encryption. A term t is new at a node $n = < s, i >$ if t is a component of $term(n)$ but t is not a component of node $< s, j >$ for every $j < i$. A component is new even if it has occurred earlier as a nested subterm of some larger component. When a component occurs new in a regular node but was a subterm of some previous node, then the principal executing that strand has done some cryptographic work to extract it as a new component[6].

2.2 The Coq Proof Assistant Overview

2.2.1 What is Coq?

We briefly describe what the Coq proof assistant is in this section.

The Coq system is a computer tool for mechanically verifying theorem proofs, and at the same time a functional programming language with a powerful type system.

Once you have proved something in Coq, you have strong assurance that it is true - more than what you usually have when doing a pen-and-paper proof. These theorems may concern usual mathematics, proof theory, or program verification. The Coq proof assistant is very powerful and expressive both for reasoning and programming. We can construct from simple terms and write simple proofs to building whole theories and complex algorithms. It provides an environment for defining objects (integers, sets, trees, functions...), making statements using logical connectives and basic predicates, and writing proofs. It also provides program extraction towards Haskell and Ocaml for efficient execution of algorithms and linking with other libraries.

The Coq compiler automatically checks the correctness of definitions (well-formed sets, terminating functions...) and of proofs [8].

As a proof assistant, Coq is similar to higher order logic (HOL) systems, a family of interactive theorem prover based on Church's HOL including Isabelle, PVS... Unlike these systems, Coq is based on intuitionistic type theory. Consequently, it is closer to Epigram, and NuPrl... The common properties of these system are that functions are programs that can be computed and not just binary relation. Coq can be used from standard teletype-like shell window but preferably through the graphical user interface called CoqIde. Coq is not an automated theorem prover which means that it does not automatically prove theorems. However, it can be considered as a semi-automated theorem prover since it includes many automatic theorem proving tatics and various decision procedures. It greatly simplifies the development of formal proofs by automating some aspects of it.

Under programming language point of view, Coq implements dependently typed functional programming language, while under logical system, it implements a higher-order type theory [3]. Coq exploits the notion of Curry-Howard isomorphism - the correspondence between proofs and programs. The relation between a proof and the statement it proves is the same as the relation between a program and its type. At the level of proofs and programs, we have the following correspondence summarized in Table 1.1.

Logic side	Programming side
hypothesis	free variables
implication elimination	application
implication introduction	abstraction

Table 2.1: At the Level of Proofs and Programs

And Table 1.2 summaries the correspondence at the level of terms and types. The

Logic side	Programming side
universal quantification	generalised function space
existential quantification	generalised cartesian product
implication	function type
conjunction	product type
disjunction	sum type
true formula	unit type
false formula	bottom (empty) type

Table 2.2: At the Level of Terms and Types

correspondence says that, for example, implication behaves the same as a function type, conjunction as product type, and disjunction as sum type. The assertion $T : \tau$ means that the term T is of type τ or equivalently that T is a proof of the proposition τ. A type $A \rightarrow B$ is the type of a function that associates a term of type B to any term of type A, while a proof of $A \rightarrow B$ is a term of that type or a term of the form $\lambda x.t$ where x is a proof of A and t is a proof of B.

There is usually a syntactic distinction between types and terms in most type theories. However, types and terms are defined as the same syntatic structure so everything even type is a term in Coq. Consequently, all objects have a type: atomic types, types for functions, types for proofs, types for types. When manipulated as terms, types are themselves a type which is a constant of the language called a sort. Prop and Set are the two base sorts. The sort Prop is the universe of propositions. The sort Set intends to be the type of small sets and includes data types such as booleans, natural numbers, and but also includes products, subsets, function type over these data types [1].

The original Coq system was based on the Calculus of Constructions (CoC). Version 7 was based on a generalization of CoC, the Calculus of Inductive Constructions (CIC). Since version V8 it is based on a weaker calculus, namely Predicate Calculus of Inductive Constructions (pCIC). The language of CIC also has typed terms, conversion rules, derived rules, and (co)inductive definitions [1].

2.2.2 Coq Architecture

Coq have two levels architecture - kernel and environment. A relatively small kernel based on a language with few primitive constructions (sorts, functions, inductive definitions, product types...) and a limited number of rules for type checking and computation. On top of the kernel, there is a rich environment to help designing theories and proofs. This environment offers mechanism like user extensible notations, tatics for proof automation, libraries... Any definition or proof defined in the environment is ultimately checked by the kernel so the environment can be used and extended safely [8].

As a Coq user, using high level constructions will help to solve a problem quickly. However, it might also important to understand the underlying low level language in order to develop new functionalities and to better control how certain constructions work.

Chapter 3

Message_Algebra

This chapter contains the formalization of the message algebra. We define the set of possible messages that can be exchanged between principals in a protocols and the relations on messages.

```
Require Import Relation_Definitions Relation_Operators
               Omega Arith ListSet FSetInterface.
```

3.1 Texts

3.1.1 Definition

```
Variable Text : Set.
```

3.1.2 Decidable equality for texts

```
Variable eq_text_dec : ∀ (x y:Text), {x = y} + {x ≠ y}.
Hint Resolve eq_text_dec.
```

3.2 Keys

Interesting design choices about keys. Here we do not model symmetric and asymmetric keys as separate types; the distinction is just different constructor/injections into the key type. Sometimes simpler. Possible issue is with key inverses...?

3.2.1 Definition

```
Variable Key : Set.
Parameter K_p : set Key.
```

3.2.2 Inverse relation for keys

```
Variable inv : relation Key.
```

3.2.3 Inv is commutative

```
Axiom inv_comm : ∀ k k', inv k k' → inv k' k.
```

3.2.4 Decidable equality for keys

```
Variable eq_key_dec : ∀ (x y:Key), {x=y} + {x≠y}.
Hint Resolve eq_key_dec.
```

3.3 Messages

In my formalization, messages are terms (as in the paper). We now define the set of messages.

3.3.1 Inductive definition for messages

```
Inductive msg : Set :=
  | T : Text → msg
  | K : Key → msg
  | P : msg → msg → msg
  | E : msg → Key → msg.
Hint Constructors msg.
```

3.3.2 Decidable equality for messages

```
Definition eq_msg_dec : ∀ x y : msg,
  {x = y} + {x ≠ y}.
Proof.
```

```
intros.
decide equality.
Qed.
Hint Resolve eq_msg_dec.
```

3.3.3 Signed messages

In a protocol, principals can either send or receive messages.We repesent transmission of a message as the occurrence of that message with positive sign, and reception of a message as its occurrence with negative sign [6]. So in Coq, signed messages are defined as an inductive set with two constructors, one for possitive signed messages and the other for negative signed messages.

Definition

```
Inductive smsg :=
  | xmit_msg : msg → smsg
  | recv_msg : msg → smsg.
Notation "+ m" := (xmit_msg m) (at level 30) : ma_scope.
Notation "- m" := (recv_msg m) : ma_scope.
```

Signed messages to messages

A function to convert signed messages to messages.

```
Definition smsg_2_msg (m : smsg) : msg :=
   match m with
   | (xmit_msg x) ⇒ x
   | (recv_msg x) ⇒ x
   end.
Hint Resolve smsg_2_msg.
```

Decidable equality for signed messages

```
Definition eq_smsg_dec : ∀ (x y : smsg), {x=y} + {x≠y}.
Proof.
intros.
decide equality.
Qed.
Hint Resolve eq_smsg_dec.
```

3.3.4 Atomic messages

A message is atomic if it is either a text message or a key message.

```
Inductive atomic : msg → Prop :=
  |atomic_text : ∀ t, atomic (T t)
  |atomic_key : ∀ k, atomic (K k).
Hint Constructors atomic.
```

3.3.5 Concatenated messages

```
Inductive pair : (msg → Prop) :=
  | pair_step : ∀ m1 m2, pair (P m1 m2).
Hint Constructors pair.
```

3.3.6 Encrypted messages

```
Inductive enc : msg → Prop :=
  | enc_step : ∀ m k, enc (E m k).
Hint Constructors enc.
```

3.3.7 Simple message

A message is simple if it is not a concatenated (paired) message.

```
Inductive simple : msg → Prop :=
  | simple_step : ∀ m, ¬ pair m → simple m.
```

Encrypted implies simple `Lemma enc_imp_simple : ∀ x k, simple (E x k).`

```
Proof.
intros x k.
apply simple_step.
unfold not. intro Hpair.
inversion Hpair.
Qed.
```

3.3.8 Some basic results about atomic, paired, and simple

```
Lemma pair_not_atomic :
  ∀ m, pair m → ¬ atomic m.
Proof.
```

```
unfold not.
intros m HConc HAtom.
inversion HConc; inversion HAtom; subst; discriminate.
Qed.
```

```
Lemma atom_not_pair:
  ∀ m, atomic m → ¬ pair m.
Proof.
unfold not.
intros m HAtom Hpair.
inversion Hpair; inversion HAtom; subst; discriminate.
Qed.
```

```
Lemma enc_not_atomic : ∀ m1 m2, ¬ atomic (P m1 m2).
Proof.
unfold not.
intros m1 m2 Hatom.
inversion Hatom.
Qed.
```

```
Lemma atomic_imp_simple : ∀ a, atomic a → simple a.
Proof.
intros a Hatom.
apply simple_step.
apply atom_not_pair; assumption.
Qed.
```

3.4 Freeness assumptions

Pair and enryption freess assumptions are provable in this context. If two concatenated (or encrypted) messages are equal then each component of the first is equal the corresponding componet of the second.

3.4.1 Pair freeness

```
Lemma pair_free : ∀ m1 m2 m1' m2',
                        P m1 m2 = P m1' m2' → m1 = m1' ∧ m2 = m2'.
Proof.
intros m1 m2 m1' m2' HPeq.
injection HPeq. auto.
Qed.
```

3.4.2 Encryption Freeness

```
Lemma enc_free : ∀ m k m' k',
                    E m k = E m' k' → m = m' ∧ k = k'.
Proof.
intros m k m' k' HEeq.
injection HEeq.
auto.
Qed.
```

3.5 Ingredients

Called "carried by" in some CPSA publications, and "subterm" in the "Authentication Tests and the structures of bundles".

3.5.1 Definition

The ingred relation is defined inductively as following.

```
Inductive ingred : msg → msg → Prop :=
| ingred_refl : ∀ m, ingred m m
| ingred_pair_l : ∀ m l r,
    ingred m l → ingred m (P l r)
| ingred_pair_r : ∀ m l r,
    ingred m r → ingred m (P l r)
| ingred_encr : ∀ m x k,
    ingred m x → ingred m (E x k).
Notation "a ¡st b" := (ingred a b) (at level 30) : ss_scope.
Open Scope ss_scope.
```

3.5.2 Proper ingredient

```
Definition proper_ingred (x y: msg) : Prop :=
  ingred x y ∧ x ≠ y.
Notation "a ¡¡st b" := (proper_ingred a b) (at level 30) : ss_scope.
```

3.5.3 Properties of the ingredient relation

Transitive

```
Lemma ingred_trans :
  ∀ x y z, x <st y → y <st z → x <st z.
Proof.
intros x y z Sxy Syz.
induction Syz.
  subst; auto.
  subst; apply ingred_pair_l; apply IHSyz; assumption.
  subst; apply ingred_pair_r; apply IHSyz; assumption.
  apply ingred_encr; apply IHSyz; assumption.
Qed.
Hint Resolve ingred_trans.
```

Some other basic results about ingredients

```
Lemma ingred_pair : ∀ (x y z:msg), x ≠ (P y z) →
                                   x <st (P y z) →
                                   x <st y ∨ x <st z.
Proof.
intros x y z Hneq Hst.
inversion Hst; subst.
  elim Hneq; trivial.
  auto.
  auto.
Qed.
Hint Resolve ingred_pair.
```

```
Lemma ingred_enc : ∀ (x y :msg) (k:Key), x ≠ (E y k) →
                                        x <st (E y k) →
                                        x <st y.
Proof.
intros x y k Hneq Hst.
inversion Hst; subst.
  elim Hneq; trivial.
  auto.
Qed.
Hint Resolve ingred_enc.
```

3.6 Size of messages

3.6.1 Definition

```
Fixpoint size (m:msg) :=
  match m with
   | T t ⇒ 1
   | K k ⇒ 1
   | P m1 m2 ⇒ (size m1) + (size m2)
   | E x k ⇒ (size x) + 1
   end.
```

Size of every message is always positive `Lemma zero_lt_size : ∀ x, 0 < size x.`

```
Proof.
intro x.
induction x; simpl; omega.
Qed.
Hint Resolve zero_lt_size.
```

```
Lemma size_lt_plus_l : ∀ x y, size x < size x + size y.
Proof.
intros x y.
assert (size x + 0 < size x + size y).
apply plus_lt_compat_l. apply zero_lt_size.
rewrite (plus_comm (size x) 0) in H.
rewrite (plus_O_n (size x)) in H. auto.
Qed.
```

3.6.2 Realtionship between ingredient and size

Size of an ingredient x is always less than or equal size of message y if x is an ingredient of y.

```
Lemma ingred_lt :
  ∀ x y, x <st y → size(x) ≤ size(y).
Proof.
intros x y Hst.
induction Hst; subst; simpl; omega.
Qed.
```

```
Lemma ingred_ge_size_eq :
  ∀ x y, x <st y → size(x) ≥ size(y) → x=y.
Proof.
intros x y Hst Hsize_gt.
```

```
inversion Hst; subst.
  auto.
  assert (Hx_lt_l : size x ≤ size l). apply ingred_lt; auto.
   assert (Hl_lt_Plr : size l < size (P l r)).
     simpl. apply size_lt_plus_l.
   assert (Hx_lt_Plr : size x < size (P l r)). omega.
   contradict Hsize_gt. omega.
  assert (Hx_lt_r : size x ≤ size r). apply ingred_lt; auto.
   assert (Hl_lt_Plr : size r < size (P l r)).
     simpl. rewrite ← (plus_comm). apply size_lt_plus_l.
   assert (Hx_lt_Plr : size x < size (P l r)). omega.
   contradict Hsize_gt. omega.
  assert (Hx_lt_l : size x ≤ size x0). apply ingred_lt; auto.
   assert (Hx0_lt_E : size x0 < size (E x0 k)).
     simpl. omega.
   assert (Hx_lt_E : size x < size (E x0 k)). omega.
   contradict Hsize_gt. omega.
Qed.
Hint Resolve ingred_ge_size_eq.
```

If each message is an ingredient of each other, then they are equal.

```
Lemma ingred_eq : ∀ (x y :msg), x <st y → y <st x → x = y.
Proof.
intros x y Hxy Hyx.
apply ingred_ge_size_eq.
auto.
apply ingred_lt; auto.
Qed.
Hint Resolve ingred_eq.
```

```
Lemma atomic_ingred_eq :
   ∀ x a, atomic a → ingred x a → x=a.
Proof.
intros x a Hat Hin.
inversion Hat; subst; inversion Hin; auto.
Qed.
Hint Resolve atomic_ingred_eq.
```

3.7 Components

Intuitively, a message x is a component of a message m if we can get x just by seperation out all the pairs in m, without using decryption.

3.7.1 Component of a message

A message t0 is an e-ingredients of message t if t is in the smallest set containing t0 and closed under concatenation with arbitrary term t1, i.e, if t0 is an atomic value of t.

```
Inductive e_ingred : relation msg :=
  | e_ingred_refl : ∀ (t0:msg), e_ingred t0 t0
  | e_ingred_pair_l : ∀ t0 t1 t2,
      e_ingred t0 t1 → e_ingred t0 (P t1 t2)
  | e_ingred_pair_r : ∀ t0 t1 t2,
     e_ingred t0 t2 → e_ingred t0 (P t1 t2).
Hint Constructors e_ingred.
```

```
Inductive comp : relation msg :=
  | comp_step : ∀ m1 m2,
                    simple m1 → e_ingred m1 m2 → comp m1 m2.
Notation ”a ¡com b” := (comp a b) (at level 30) : ss_scope.
Hint Constructors comp.
```

3.7.2 Component implies ingredient

```
Lemma e_ingred_imp_ingred : ∀ m1 m2, e_ingred m1 m2 → ingred m1 m2.
Proof.
intros m1 m2 Hein.
induction Hein; subst.
  apply ingred_refl.
  apply ingred_pair_l; assumption.
  apply ingred_pair_r; assumption.
Qed.
```

```
Lemma comp_imp_ingred : ∀ (m1 m2:msg), m1 <com m2 → m1 <st m2.
Proof.
intros m1 m2 Hcom.
apply e_ingred_imp_ingred.
inversion Hcom; subst; assumption.
Qed.
```

3.7.3 Concatenation or pairing preserves components

If a message x is a component an other message m1, it also is a component of every message which is concatenated from m1 and an abitrary message m2 Lemma preserve_comp_l : ∀ x $m1$ $m2$, **comp** x $m1$ → **comp** x (P $m1$ $m2$).

```
Proof.
intros x m1 m2 Hcom.
inversion Hcom; subst.
apply comp_step.
 auto.
 apply e_ingred_pair_l; assumption.
Qed.
```

Lemma preserve_comp_r : ∀ x $m1$ $m2$, **comp** x $m2$ → **comp** x (P $m1$ $m2$).

```
Proof.
intros x m1 m2 Hcom.
inversion Hcom; subst.
apply comp_step.
 auto.
 apply e_ingred_pair_r; assumption.
Qed.
```

3.7.4 An atomic message is a component of itself

Lemma comp_atomic_cyclic : ∀ a, **atomic** a → **comp** a a.

```
Proof.
intros a Hatom.
constructor.
  apply atomic_imp_simple; assumption.
  apply e_ingred_refl.
Qed.
```

3.7.5 A simple message is a component of itself

Lemma comp_simple_cyclic : ∀ a, **simple** a → **comp** a a.

```
Proof.
intros a Hsim.
constructor; [assumption |constructor].
Qed.
```

3.8 K-ingredients

```
Section K_relation.
```

A message t0 is an k-ingredients of message t if t is in the smallest set containing t0 and closed under encryption and concatenation with arbitrary term t1, i.e, if t0 is an atomic value of t.

```
Variable F : Set.
Parameter inj_F_K : F → Key.
Axiom inj_F_K_inj : ∀ x y : F, inj_F_K x = inj_F_K y → x = y.
Coercion inj_F_K : F ¿-¿ Key.
Inductive k_ingred : relation msg :=
  | k_ingred_refl : ∀ (t0:msg), k_ingred t0 t0
  | k_ingred_pair_l : ∀ (t0 t1 t2 : msg),
       k_ingred t0 t1 → k_ingred t0 (P t1 t2)
  | k_ingred_pair_r : ∀ (t0 t1 t2 : msg),
      k_ingred t0 t2 → k_ingred t0 (P t1 t2)
  | k_ingred_enc : ∀ (t0 t1 : msg) (k : F),
      k_ingred t0 t1 → k_ingred t0 (E t1 k).
Hint Constructors k_ingred.
End K_relation.
```

Chapter 4

Strand_Spaces

This chapter contains the formalization of most of the basic concepts of strand spaces, including strand, node, penetrator strand, strand edges, new component...

```
Require Import Classical.
Require Import Message_Algebra.
Require Import Lists.ListSet Lists.List.
Require Import Omega ZArith.
Require Import Relation_Definitions Relation_Operators.
```

```
Open Scope list_scope.
Import ListNotations.
Open Scope ma_scope.
```

4.1 Strands

4.1.1 Strand Definition

A strand is a sequence of events; it represents either an execution by a legitimate party in a security protocol or else a sequence of actions by a penetrator [6]. In Coq, we define a strand as a list of signed messages.

```
Definition strand : Type := list smsg.
```

4.1.2 Decidable equality for strands

It is provable in this context.

```
Definition eq_strand_dec : ∀ x y : strand,{x = y} + {x ≠ y}.
```

```
Proof.
intros. decide equality.
Qed.
Hint Resolve eq_strand_dec.
```

4.2 Nodes

4.2.1 Definition

A node is a pair of a strand and a natural number, which is less than the length of the strand. The natural number is called "index" of that node. Note that the list index in Coq starts from zero.

```
Definition node : Type := {n:(prod strand nat) | snd n < length (fst n)}.
```

4.2.2 Strand of a node

Strand of a node function takes a node and returns the strand of that node.

```
Definition strand_of (n:node) : strand := match n with
  | exist apair _ ⇒ fst apair end.
```

4.2.3 Index of a node

Index of a node function takes a node and returns the index of that node.

```
Definition index_of (n:node) : nat := match n with
  | exist apair _ ⇒ snd apair end.
```

4.2.4 Decidable equality for nodes

For any two nodes, we can decide whether they are equal or not.

```
Definition eq_node_dec : ∀ x y : node,
 {x = y} + {x ≠ y}.
Proof.
  intros [[xs xn] xp] [[ys yn] yp].
  destruct (eq_strand_dec xs ys) as [EQs | NEQs]; subst.
  destruct (eq_nat_dec xn yn) as [EQn | NEQn]; subst.
  left. rewrite (proof_irrelevance (lt yn (length ys)) xp yp). reflexivity.
  right. intros C. inversion C. auto.
  right. intros C. inversion C. auto.
Qed.
Hint Resolve eq_node_dec.
```

4.2.5 Signed message of a node

We want to have a function that takes a node and returns the signed message of that node. However, it is a little bit hard to write it in Coq since node is a dependent type. Specificly, a node just contains its strand and its index, so we need to extract the signed message at the "index-th" position on the strand. Below are some helper functions for defining such the function.

```
Definition option_smsg_of (n:node) : (option smsg) :=
  match n with
  | exist (s,i) _ ⇒ nth_error s i end.
Lemma nth_error_len :
  ∀ (A:Type) (l:list A) (n:nat),
    nth_error l n = None → (length l) ≤ n.
Proof.
intros A l n. generalize dependent l.
induction n.
intros l H.
unfold nth_error in H.
unfold error in H. destruct l. auto. inversion H.
intros l1 H. destruct l1. simpl; omega.
inversion H. apply IHn in H. simpl. omega.
Qed.
Lemma valid_smsg : ∀ (n:node), {m:smsg | option_smsg_of n = Some m}.
Proof.
intros n.
remember (option_smsg_of n) as opn.
destruct n. destruct opn.
∃ s. auto.
unfold option_smsg_of in Heqopn.
destruct x. simpl in l.
symmetry in Heqopn.
apply nth_error_len in Heqopn.
omega.
Qed.
```

Here is the actual signed message of a node function.

```
Definition smsg_of (n:node) : smsg := match (valid_smsg n) with
  | exist m _ ⇒ m end.
```

4.2.6 Unsigned message of a node

To get the unsigned message of a node, just convert its singed message to the unsigned one.

```
Definition msg_of (n:node) : msg := smsg_2_msg (smsg_of n).
```

4.2.7 Predicate for positive and negative nodes

A node is a positive (transmission) node if the signed message of that node is positive

```
Definition xmit (n:node) : Prop := ∃ (m:msg), smsg_of n = + m.
```

A node is a negative (reception) node if the signed message of that node is negative

```
Definition recv (n:node) : Prop := ∃ (m:msg), smsg_of n = - m.
```

4.3 Penetrator Strands

```
Section PenetratorStrand.
```

The penetrator's powers are characterized by the set of compromised keys which are initially known to penetrator, and a set of penetrator strands that allow the penetrator to generate new messages. The set of compromised keys typically would contain all public keys, all private keys of penetrators, and all symmetric keys initially shared between the penetrator and principals playing by the protocol rules [9].

```
  Parameter K_p : set Key.
```

The atomic actions available to penetrator are encoded in a set of penetrator strands. We partition penetrator strands according to the operations they exemplify.

4.3.1 Text Message Strand

M-strands emit known atomic text or guess.

```
Inductive MStrand (s : strand) : Prop :=
| P_M : ∀ t : Text, s = [+ (T t)] → MStrand s.
Hint Constructors MStrand.
```

4.3.2 Key Strand

K-strands emit keys from a set of known keys.

```
Inductive KStrand (s : strand) : Prop :=
| P_K : ∀ k : Key, set_In k K_p → s = [+ (K k)] → KStrand s.
Hint Constructors KStrand.
```

4.3.3 Concatenation Strand

C-strands concatenate terms.

```
Inductive CStrand (s : strand) : Prop :=
| P_C : ∀ (g h : msg), s = [- g; - h; + (P g h)] → CStrand s.
Hint Constructors CStrand.
```

4.3.4 Separation Strand

S-strands separate terms.

```
Inductive SStrand (s : strand) : Prop :=
| P_S : ∀ (g h : msg), s = [- (P g h); + g ; + h] → SStrand s.
Hint Constructors SStrand.
```

4.3.5 Encryption Strand

E-strands encrypt when given a key and a plain-text.

```
Inductive EStrand (s : strand) : Prop :=
| P_E : ∀ (k : Key) (h :msg), s = [- (K k); - h; + (E h k)] → EStrand s.
Hint Constructors EStrand.
```

4.3.6 Decryption Strand

D-strands decrypt when given a decryption key and matching cipher-text.

```
Inductive DStrand (s : strand) : Prop :=
| P_D : ∀ (k k' : Key) (h :msg),
  inv k k' → s = [- ( K k'); - (E h k); + h] → DStrand s.
Hint Constructors DStrand.
```

4.3.7 Definition for PenetratorStrand

Hence, a strand is called a penetrator strand if it is one of the above strands.

```
Inductive PenetratorStrand (s:strand) :Prop :=
| PM : MStrand s → PenetratorStrand s
| PK : KStrand s → PenetratorStrand s
| PC : CStrand s → PenetratorStrand s
| PS : SStrand s → PenetratorStrand s
| PE : EStrand s → PenetratorStrand s
| PD : DStrand s → PenetratorStrand s.
Hint Constructors PenetratorStrand.
```

4.3.8 Predicates for penetrable nodes and regular nodes

A node is a penetrator node if the strand it lies on is a penetrator strand.

```
Definition p_node (n:node) : Prop := PenetratorStrand (strand_of(n)).
```

A non-penetrator node is called a regular node.

```
Definition r_node (n:node) : Prop := ¬ p_node n.
```

4.3.9 Axiom for penetrator node and regular node

Every node is either a penatrator node or regular node.

```
Axiom node_p_or_r : ∀ (n:node), p_node n ∨ r_node n.
```

```
End PenetratorStrand.
```

4.4 Edges

4.4.1 Inter-strand Edges

The inter-strand communication is represented as a relation on nodes. x –¿ y means that a transmission node x sends message to a reception node y.

Inductive **msg_deliver** : relation node :=
| msg_deliver_step : ∀ (x y : node) (m:**msg**),
smsg_of x = +m ∧ smsg_of y = -m ∧ strand_of(x) ≠ strand_of(y)
→ **msg_deliver** x y.
Hint Constructors **msg_deliver**.
Notation "x –¿ y" := (**msg_deliver** x y) (at level 0, right associativity) : *ss_scope*.

4.4.2 Iner-strand Edges - Strand ssuccessor

A node y is the successor of a node x, denoted as x ==¿ y, if they are on the same strand and y is immediately after x on the list of nodes of the strand.

Inductive **ssucc** : relation node :=
| ssucc_step : ∀ (x y : node), strand_of(x) = strand_of(y) ∧
index_of(x) + 1 = index_of(y) → **ssucc** x y.
Hint Constructors **ssucc**.
Notation "x ==¿ y" := (**ssucc** x y) (at level 0, right associativity) : *ss_scope*.

Transitive closure of strand ssuccessor Definition ssuccs : relation node := **clos_trans** node **ssucc**.
Notation "x ==¿+ y" := (ssuccs x y) (at level 0, right associativity) : *ss_scope*.

Reflexive Transitive Closure of strand successor Definition ssuccseq : relation node := **clos_refl_trans** node **ssucc**.

4.4.3 Edges on Strand

An edge is a realtion on nodes and it is either a inter-strand or inner-strand relation.

Inductive **strand_edge** : relation node :=
| strand_edge_single : ∀ x y, **msg_deliver** x y → **strand_edge** x y
| strand_edge_double : ∀ x y, **ssucc** x y → **strand_edge** x y.
Hint Constructors **strand_edge**.

Transitive closure of edge Definition prec := **clos_trans** node **strand_edge**.
Notation "x ==¿* y" := (ssuccseq x y) (at level 0, right associativity) : *ss_scope*.

4.4.4 Constructive and Destructive Edges

An edge is constructive if both nodes lie on a encryption or concatenation strand.

```
Inductive cons_edge : relation node :=
  | cons_e : ∀ x y, ssuccs x y → EStrand (strand_of x) → cons_edge x y
  | cons_c : ∀ x y, ssuccs x y → CStrand (strand_of x) → cons_edge x y.
Hint Constructors cons_edge.
```

An edge is destructive if both nodes lie on a decryption or separation strand.

```
Inductive des_edge : relation node :=
  | des_d : ∀ x y, ssuccs x y → DStrand (strand_of x) → des_edge x y
  | des_s : ∀ x y, ssuccs x y → SStrand (strand_of x) → des_edge x y.
Hint Constructors des_edge.
```

4.5 Origination

We say that a message m is originate at a node n if n is a trasmission node, m is an ingredient of the message of n, and m is not an ingredient of any earlier node of n.

```
Definition orig_at (n:node) (m:msg) : Prop :=
  xmit(n) ∧ (ingred m (msg_of n)) ∧
  (∀ (n':node), ((ssuccs n' n) →
  (ingred m (msg_of n')) → False)).
Definition non_orig (m:msg) : Prop := ∀ (n:node), ¬orig_at n m.
```

If a value originates on only one node in the strand space, we call it uniquely originating.

```
Definition unique (m:msg) : Prop :=
  (∃ (n:node), orig_at n m) ∧
  (∀ (n n':node),(orig_at n m) ∧ (orig_at n' m) → n=n').
```

4.6 Axioms

4.6.1 The bundle axiom: every received message was sent

```
Axiom was_sent : ∀ x : node, (recv x) →
  (∃ y : node, msg_deliver y x).
```

4.6.2 Normal bundle axiom

```
Axiom not_k_k : ∀ k k', inv k k' → DStrand [-(K k); -(E (K k) k'); + (K k)].
```

4.6.3 Well-foundedness

```
Axiom wf_prec: well_founded prec.
```

4.7 Minimal nodes

```
Definition is_minimal: (node → Prop) → node → Prop :=
  fun P x ⇒ (P x) ∧ ∀ y, (prec y x) → ~( P y).
Definition has_min_elt: (node → Prop) → Prop :=
  fun P ⇒ ∃ x:node, is_minimal P x.
```

4.8 New Component

4.8.1 Component of a node

A message is a component of a node if it is a component of the message at that node.

```
Definition comp_of_node (m:msg) (n:node) : Prop := comp m (msg_of n).
Notation "x ¡[node] y" := (comp_of_node x y) (at level 50) : ss_scope.
```

4.8.2 New at

A message is new at a node if it is a component of that node and the message is not a component of any ealier node in the same strand with the node.

```
Definition new_at (m:msg) (n:node) : Prop :=
  m <[node] n ∧ ∀ (n' : node) , ssuccs n' n → m <[node] n'→ False.
```

4.9 Paths

```
Section Path.
  Parameter default_node : node.
```

4.9.1 Path condition

A path_edge is either a message deliver or a ssuccs where the first node is positive and the second node is negative.

```
Inductive path_edge (m n : node) : Prop :=
| path_edge_single : msg_deliver m n → path_edge m n
| path_edge_double : ssuccs m n ∧ recv(m) ∧ xmit(n) → path_edge m n.
Hint Constructors path_edge.
Notation ”m —–¿ n” := (path_edge m n) (at level 30) : ss_scope.
```

4.9.2 The n-th node of a path

It takes a natural number and a list of nodes and returns the node at the n-th postition on the list.

```
Definition nth_node (i:nat) (p:list node) : node :=
  nth_default default_node p i.
Hint Resolve nth_node.
```

4.9.3 Definitions for paths

A path is any finite sequence of nodes where for all two consecutive nodes they form a path edge.

```
Definition is_path (p:list node) : Prop :=
  ∀ i, i < length(p) - 1 → path_edge (nth_node i p) (nth_node (i+1) p).
```

4.9.4 Axiom for paths

All paths begin on a positive node and end on a negative node.

```
Axiom path_begin_pos_end_neg : ∀ (p:list node),
  xmit(nth_node 0 p) ∧ recv(nth_node (length(p)-1) p).
```

4.9.5 Penetrator Paths

A penetrator path is one in which all nodes other than possibily the first or the last are pentrator nodes.

```
Definition p_path (p:list node): Prop := is_path p ∧ ∀ i,
  (i > 0 ∧ i < length p - 1) → p_node (nth_node i p).
```

Any penetrator path that begins at a regular node contains only constructive and destructive edges.

```
Lemma p_path_cons_or_des :
  ∀ p, p_path p → r_node (nth_node 0 p) →
  (∀ i, i < length p - 1 →
  cons_edge (nth_node i p) (nth_node (i+1) p) ∨
  des_edge (nth_node i p) (nth_node (i+1) p)).
Admitted.
```

4.9.6 Falling and rising paths

A pentrator path is falling if for all adjacent nodes n, n' on the path the message of n' is an ingredient of $n's$.

```
  Definition falling_path ( p : list node) : Prop :=
    p_path p ∧ ∀ i, i < length(p)-1 →
    ingred (msg_of (nth_node (i+1) p)) (msg_of (nth_node i p)).
```

A pentrator path is rising if for all adjacent nodes n, n' on the path the message of n is an ingredient of the message of n'.

```
  Definition rising_path (p : list node) : Prop :=
    p_path p ∧ ∀ i, i < length(p)-1 →
    ingred (msg_of (nth_node i p)) (msg_of (nth_node (i+1) p)).
```

4.9.7 Destructive and Constructive Paths

A penetrator path is constructive if it contains only constructive edges.

```
  Definition cons_path (p :list node) : Prop :=
    p_path p ∧ (∀ i, i < length p - 1 →
                    ssuccs (nth_node i p) (nth_node (i+1) p) →
                    cons_edge (nth_node i p) (nth_node (i+1) p)).
  Definition cons_path_not_key (p : list node) : Prop :=
    cons_path p ∧ (∀ i, i < length p - 1 →
    des_edge (nth_node i p) (nth_node (i+1) p) →
    EStrand (strand_of (nth_node i p)) →
    ∃ k , msg_of (nth_node i p) = K k → False).
```

A penetrator path is destructive if it contains only destructive edges.

```
  Definition des_path (p :list node) : Prop :=
    p_path p ∧ (∀ i, i < length p - 1 →
                ssuccs (nth_node i p) (nth_node (i+1) p) →
                des_edge (nth_node i p) (nth_node (i+1) p)).
Definition des_path_not_key (p : list node) : Prop :=
  des_path p ∧ (∀ i, i < length p - 1 →
  des_edge (nth_node i p) (nth_node (i+1) p) →
  DStrand (strand_of (nth_node i p)) →
  ∃ k, msg_of (nth_node i p) = K k → False).
End Path.
```

4.10 Penetrable Keys and Safe Keys

Penetrable key is already penetrated (K_p) or some regular strand puts it in a form that could allow it to be penetrated, because for each key protecting it, the matching key decryption key is already pentrable [6].

```
Section Penetrable_Keys.
  Parameter Kp : Set.
  Parameter Pk : nat → Key → Prop.
  Axiom init_pkeys : sig (Pk 0) = Kp.
  Axiom next_pkeys : ∀ (i:nat) (k:Key), (∃ (n:node) (t:msg),
    r_node n ∧ xmit n ∧ new_at t n ∧
    k_ingred (sig (Pk i)) (K k) t) → Pk (i+1) k.
  Inductive PKeys (k:Key) : Prop :=
  | pkey_step : (∃ (i:nat), Pk i k) → PKeys k.
End Penetrable_Keys.
```

4.11 Transformation paths

Given a test of the form $n \Rightarrow^{+} n'$, the strategy for proving the authentication test results is to consider the paths leading from n to n'. Because there is a value a originating uniquely at n, and it is received back at n', there must be a path leading from n to n'(apart from the trivial path that follows the strand from n to n'). Moreover, since a is received in a new form at n', there must be a step along the path that changes its form; this is a transforming edge. The incoming and outgoing authentication test results codify conditions under which we can infer that a transforming edge lies on a regular strand [6].

The proofs focus on the transformation paths leading from n to n' that keep track of a relevant component containing a. The relevant component changes only when a transforming edge is traversed, and a occurs in a new component of a node between n and n'. We regard the edge $n \Rightarrow^{+} n'$ as a transformed edge, because the same value a occurs in both nodes, but node n contains a in transformed form[1]. Notice that the difinition of transformed and transforming edges are modified a little bit to make the proof work precisely. The component of n' containing a is not necessarily new at n' but it is new at some node in between n and n' [6].

```
Section Trans_path.
  Definition path : Type := list (prod node msg).
  Variable p : path.
  Variable a : msg.
  Parameter default_msg : msg.
  Definition ln := fst (split p).
  Hint Resolve ln.
  Definition lm := snd (split p).
  Hint Resolve lm.
```

A function that takes a natural number and a list of messages and returns the message at the n-th postion in the list. If the natural number is out of range, then a default message is returned.

```
  Definition nth_msg : nat → list msg → msg :=
    fun (n:nat) (p:list msg) ⇒ nth_default default_msg p n.
  Hint Resolve nth_msg.
  Definition L (n:nat) := nth_msg n lm.
  Hint Resolve L.
  Definition nd (n:nat) := nth_node n ln.
  Hint Resolve nd.
```

An abstract predicate for defining transforming edge and transformed edge.

```
  Definition transformed_edge (x y : node) (a:msg) : Prop :=
    ssuccs x y ∧ atomic a ∧
    ∃ z Ly, ssuccs x z ∧ ssuccseq z y ∧
    new_at Ly z ∧ a <st Ly ∧ Ly <[node] y.
```

A transformed edge emits a atomic message a and later receives in a new form.

```
  Definition transformed_edge_for (x y : node) (a :msg) : Prop :=
    transformed_edge x y a ∧ xmit x ∧ recv y.
```

A transforming edge receive a and later emits it in transformed form.

```
  Definition transforming_edge_for (x y : node) (a :msg) : Prop :=
    transformed_edge x y a ∧ recv x ∧ xmit y.
```

A transformation path is a path for which each node n_i is labelled by a component L_i of n_i in such a way that $L_i = L_{i+1}$ unless $n_i \Rightarrow n_{i+1}$ is a trans edge.

```
  Definition is_trans_path : Prop :=
    (is_path ln ∨ (ssuccs (nd 0) (nd 1) ∧ xmit (nd 0) ∧
                    xmit (nd 1) ∧ is_path (tl ln))) ∧
    atomic a ∧
    ∀ (n:nat), (n < length p → a <st (L n) ∧ (L n) <[node] (nd n)) ∧
                (n < length p - 1 → (L n = L (n+1) ∨ (L n ≠ L (n+1) →
                transformed_edge (nd n) (nd (n+1)) a))).
```

A transformation path does not traverse the key edge of a D-strand or E-strand.

```
Definition not_traverse_key : Prop :=
    ∀ i, i < length p - 1 → (DStrand (strand_of (nd i)) ∨ EStrand (strand_of (nd
i))) →
    ∃ k , msg_of (nd i) = K k → False.
End Trans_path.
```

4.11.1 Axiom about penetrator strands and penetrator nodes

```
  Lemma P_node_strand :
    ∀ (n:node), p_node n → PenetratorStrand (strand_of n).
  Proof.
  intros n Pn. auto.
  Qed.
```

Chapter 5

Strand_Library

This chapter contains a collection of technical results conveninet for proving larger results about strand spaces.

```
Require Import Lists.List Omega Ring ZArith.
Require Import Strand_Spaces Message_Algebra.
Require Import ProofIrrelevance Classical.
Require Import Relation_Definitions Relation_Operators.
Require Import List_Library.
```

```
Import ListNotations.
```

5.1 Messages

Convert signed messages to (unsigned) messages

```
Lemma smsg_2_msg_xmit : ∀ n m, smsg_of n = +m → msg_of n = m.
Proof.
intros. unfold msg_of. rewrite H. auto.
Qed.
```

```
Lemma smsg_2_msg_recv : ∀ n m, smsg_of n = -m → msg_of n = m.
Proof.
intros. unfold msg_of. rewrite H. auto.
Qed.
```

```
Lemma node_smsg_msg_xmit : ∀ n t,
smsg_of(n) = (+ t) →
msg_of(n) = t.
Proof.
  intros n t H.
  unfold msg_of. rewrite H. reflexivity.
Qed.
Hint Resolve node_smsg_msg_xmit.

Lemma node_smsg_msg_recv : ∀ n t,
smsg_of(n) = (- t) →
msg_of(n) = t.
Proof.
  intros n t H.
  unfold msg_of. rewrite H. reflexivity.
Qed.
Hint Resolve node_smsg_msg_recv.

Lemma nth_error_some_In {X:Type}: ∀ l i (x:X),
nth_error l i = Some x →
List.In x l.
Proof.
  intros l. induction l.
  intros i x nth. destruct i. simpl in nth; inversion nth.
  simpl in nth; inversion nth.
  intros i x nth.
  destruct i. simpl in nth. inversion nth. left. reflexivity.
  simpl in nth. right. eapply IHl. exact nth.
Qed.
Hint Resolve nth_error_some_In.

Lemma nth_error_node : ∀ n,
nth_error (strand_of n) (index_of n) = Some (smsg_of n).
Proof.
  intros n.
  unfold smsg_of. destruct valid_smsg.
  destruct n. simpl in *.
  destruct x0. simpl. auto.
Qed.
Hint Resolve nth_error_node.

Lemma strand_node : ∀ (s: strand) (i: nat),
i < length s →
∃ n, strand_of n = s ∧ index_of n = i.
Proof.
  intros s i len.
  eexists (exist _ (s,i) _). simpl.
  split; reflexivity.
  Grab Existential Variables.
  simpl. exact len.
Qed.
```

```
Hint Resolve strand_node.
```

Every signed message of a node must be some signed message in the node's strand

```
Lemma smsg_in_strand : ∀ n s,
(strand_of n) = s →
List.In (smsg_of n) s.
Proof.
  intros.
  eapply nth_error_some_In. subst.
  apply nth_error_node.
Qed.
```

5.2 Xmit and recv

No node is both transmit and receive.

```
Lemma xmit_vs_recv: ∀ (n:node), xmit(n) → recv(n) → False.
Proof.
intros n Hx Hr.
inversion Hx. inversion Hr.
rewrite H in H0. discriminate.
Qed.
```

every node is either transmit or receive

```
Lemma xmit_or_recv: ∀ (n: node), xmit n ∨ recv n.
Proof.
intros n. unfold xmit, recv. case (smsg_of n).
intros. left. ∃ m. auto.
intros. right. ∃ m. auto.
Qed.
```

```
Lemma eq_nodes : ∀ (x y : node), strand_of(x) = strand_of(y) →
  index_of(x) = index_of(y) → x = y.
Proof.
  intros [[xs xn] xp] [[ys yn] yp] eq_index eq_strand.
  simpl in eq_index, eq_strand. subst.
  rewrite (proof_irrelevance (lt yn (length ys)) xp yp). reflexivity.
Qed.
```

5.3 Predecessor and message deliver

5.3.1 Baby result about msg_deliver

```
Lemma msg_deliver_xmit : ∀ x y, msg_deliver x y → xmit x.
Proof.
intros x y Md.
destruct Md.
unfold xmit. ∃ m; apply H.
Qed.
```

```
Lemma msg_deliver_recv : ∀ x y, msg_deliver x y → recv y.
Proof.
intros x y Md.
destruct Md.
unfold recv. ∃ m; apply H.
Qed.
```

5.3.2 Baby results about prec

```
Theorem prec_transitive:
  ∀ x y z, (prec x y) → (prec y z) → (prec x z).
Proof.
  apply t_trans.
Qed.
```

```
Lemma deliver_prec:
  ∀ x y, (msg_deliver x y) → (prec x y).
Proof.
  intros. constructor. constructor. auto.
Qed.
```

5.4 Succsessor

This section contains lemmas about successor, transitive closure, reflexive transitive closure, and the relations between succsessor and index of nodes. For example, if y is a successor of x, then the index of y is greater than the index of x.

```
Lemma ssucc_index_lt :
  ∀ x y, ssucc x y → index_of x < index_of y.
Proof.
intros x y Sxy.
```

```
inversion Sxy. omega.
Qed.

Lemma ssuccs_index_lt :
  ∀ x y, ssuccs x y → index_of x < index_of y.
Proof.
intros x y Sxy.
induction Sxy. apply ssucc_index_lt. auto.
omega.
Qed.

Lemma ssuccseq_index_lteq :
  ∀ x y, ssuccseq x y → index_of x ≤ index_of y.
Proof.
intros x y Sxy.
induction Sxy. assert (index_of x < index_of y).
apply ssucc_index_lt. auto. omega. auto. omega.
Qed.

Lemma index_lt_one_ssucc :
  ∀ x y, strand_of x = strand_of y → index_of x + 1= index_of y → ssucc x y.
Proof. auto. Qed.

Lemma index_lt_ssuccs :
  ∀ x y, strand_of x = strand_of y → index_of x < index_of y → ssuccs x y.
Proof.
intros x y Sxy lt.
Admitted.

Lemma ssuccs_imp_ssuccseq :
  ∀ x y, ssuccs x y → ssuccseq x y.
Proof.
intros x y Sxy.
induction Sxy. apply rt_step. auto.
apply rt_trans with (y:=y); auto.
Qed.

Lemma index_lteq_ssuccseq :
  ∀ x y, strand_of x = strand_of y → index_of x ≤ index_of y → ssuccseq x y.
Proof.
intros x y Sxy lt.
assert (index_of x = index_of y ∨ index_of x < index_of y). omega.
case H. intros. assert (x=y). apply eq_nodes; auto.
rewrite H1. apply rt_refl.
intros. apply ssuccs_imp_ssuccseq.
apply index_lt_ssuccs; auto.
Qed.
```

Strand-successor is irreflexive.

```
Lemma ssucc_acyclic: ∀ (n:node), ssucc n n → False.
Proof.
intros n Hs. inversion Hs. destruct H. omega.
Qed.
```

Transitive closure of strand successor is also irreflexive.

```
Lemma ssuccs_acyclic : ∀ (n:node), ssuccs n n → False.
Proof.
intros n Snn.
assert (index_of n < index_of n). apply ssuccs_index_lt.
auto. omega.
Qed.
```

Strand-successors are unique.

```
Lemma ssucc_unique:
  ∀ (x y z: node), ssucc x y → ssucc x z → y = z.
Proof.
  intros x y z Hxy Hxz.
  destruct Hxy, Hxz.
  apply eq_nodes; destruct H, H0; try omega; congruence.
Qed.
Hint Resolve ssucc_unique.
```

Every node and its successor are on the same strand.

```
Lemma ssucc_same_strand :
  ∀ (x y : node), ssucc x y → strand_of(x) = strand_of(y).
Proof.
intros x y Sxy. inversion Sxy. destruct H; auto.
Qed.
Hint Resolve ssucc_same_strand.

Lemma ssuccs_same_strand :
  ∀ (x y : node), ssuccs x y → strand_of x = strand_of y.
Proof.
  intros x y Sxy.
  induction Sxy.
  auto. congruence.
Qed.
Hint Resolve ssuccs_same_strand.

Lemma ssuccseq_same_strand :
  ∀ (x y : node), ssuccseq x y → strand_of x = strand_of y.
Proof.
  intros x y Sxy.
  induction Sxy.
  apply ssucc_same_strand. auto.
  auto. congruence.
Qed.
```

Successor reverses prec

```
Lemma ssucc_prec:
  ∀ x y, (ssucc x y) → (prec x y).
Proof.
  intros. constructor. apply strand_edge_double. auto.
Qed.
```

Successor implies prec.

```
Lemma ssuccs_prec:
  ∀ x y, (ssuccs x y) → (prec x y).
Proof.
  intros x y Sxy.
  induction Sxy.
  apply ssucc_prec; auto.
  apply prec_transitive with (y:=y); auto.
Qed.
```

Ssuccs is transitive

```
Lemma ssuccs_trans :
  ∀ x y z, ssuccs x y → ssuccs y z → ssuccs x z.
Proof.
intros x y z Sxy Syz.
apply t_trans with (y:=y); auto.
Qed.
```

```
Lemma path_edge_prec :
  ∀ x y, path_edge x y → prec x y.
Proof.
  intros x y Pxy.
  inversion Pxy.
  apply deliver_prec; auto.
  apply ssuccs_prec; apply H.
Qed.
```

5.5 Basic Results for Penetrator Strands

```
Lemma strand_1_node : ∀ n x, strand_of n = [x] → smsg_of n = x.
Proof.
intros n x Snx.
assert (H : List.In (smsg_of n) [x]).
apply smsg_in_strand; auto.
elim H; auto.
intro. elim H0.
Qed.
```

If n is a node of a MStrand or KStrand, then n is a positive node

```
Lemma MStrand_xmit_node :
  ∀ (n:node), MStrand (strand_of n) → xmit n.
Proof.
  unfold xmit.
  intros n Ms. inversion Ms. ∃ (T t).
  apply strand_1_node. auto.
Qed.
```

```
Lemma KStrand_xmit_node :
  ∀ (n:node), KStrand (strand_of n) → xmit n.
Proof.
  unfold xmit.
  intros n Ms. inversion Ms. ∃ (K k).
  apply strand_1_node. auto.
Qed.
```

If n is a node of a strand of lenght 3, the singed message of n is one of the 3 messages on the strand.

```
Lemma strand_3_nodes :
  ∀ n x y z, strand_of n = [x;y;z] →
  smsg_of n = x ∨ smsg_of n = y ∨ smsg_of n = z.
Proof.
  intros n x y z Sxyz.
  assert (Lxyz : List.In (smsg_of n) [x;y;z]) .
  apply smsg_in_strand; auto.
  elim Lxyz. auto.
  intro Lyz. elim Lyz; auto.
  intro Lz. elim Lz; auto.
  intro Le. elim Le; auto.
Qed.
```

A function to extract the singed message of a positive node which lies on a strand of lenght 3 including only one positive node. Lemma strand_3_nodes_nnp_xmit :

```
 ∀ n x y z, strand_of n = [-x;-y;+z] → xmit n → smsg_of n = +z.
Proof.
intros n x y z Sxyz Xn.
assert (Hxyz : smsg_of n = -x ∨ smsg_of n = -y ∨ smsg_of n = +z).
apply strand_3_nodes. auto.
case Hxyz. intro. apply False_ind. apply (xmit_vs_recv n).
  auto. unfold recv; auto; ∃ x; auto.
intros Hyz. case Hyz. intro. apply False_ind. apply (xmit_vs_recv n).
  auto. unfold recv; auto; ∃ y; auto.
auto.
Qed.
```

A function to extract the singed message of a negative node which lies on a strand of lenght 3.

```
Lemma strand_3_nodes_nnp_recv :
 ∀ n x y z, strand_of n = [-x;-y;+z] → recv n →
 smsg_of n = -x ∨ smsg_of n = -y.
Proof.
intros n x y z Sxyz Xn.
assert (Hxyz : smsg_of n = -x ∨ smsg_of n = -y ∨ smsg_of n = +z).
apply strand_3_nodes. auto.
case Hxyz. intro. left; auto.
intro Hyz. case Hyz. right; auto.
intro Hz. apply False_ind. apply (xmit_vs_recv n).
  unfold xmit. ∃ z ; auto. auto.
Qed.
```

A function to extract the singed message of a negative node which lies on a strand of lenght 3 including only one negative node.

```
Lemma strand_3_nodes_npp_recv :
 ∀ n x y z, strand_of n = [-x;+y;+z] → recv n →
 smsg_of n = -x.
Proof.
intros n x y z Sxyz Xn.
assert (Hxyz : smsg_of n = -x ∨ smsg_of n = +y ∨ smsg_of n = +z).
apply strand_3_nodes. auto.
case Hxyz. intro. auto.
intro Hyz. case Hyz. intro. apply False_ind. apply (xmit_vs_recv n).
  unfold xmit. ∃ y. auto. auto.
intro Hz. apply False_ind. apply (xmit_vs_recv n).
  unfold xmit. ∃ z ; auto. auto.
```

```
Qed.

Lemma pair_not_ingred_comp_l : ∀ x y, ¬(P x y) <st x.
Proof.
  intros x y Hingred.
  assert (Hlt : size (P x y ) ≤ size x).
  apply ingred_lt. auto.
  assert (Hgt : size (P x y) > size x).
  simpl. apply size_lt_plus_l. omega.
Qed.

Lemma pair_not_ingred_comp_r :
  ∀ x y, ¬(P x y) <st y.
Proof.
  intros x y Hst.
  assert (Hlt : size (P x y ) ≤ size y).
  apply ingred_lt. auto.
  assert (Hgt : size (P x y) > size y).
  simpl. rewrite (plus_comm (size x) (size y)).
  apply size_lt_plus_l. omega.
Qed.

Lemma enc_not_ingred_comp_l : ∀ x y, ¬(E x y) <st x.
Proof.
  intros x y Hingred.
  assert (Hlt : size (E x y ) ≤ size x).
  apply ingred_lt. auto.
  assert (Hgt : size (E x y) > size x).
  simpl. omega. omega.
Qed.

Lemma enc_not_ingred_comp_r :
  ∀ x y, ¬(E x y) <st (K y).
Proof.
  intros x y Hst.
  assert (Hlt : size (E x y ) ≤ size (K y)).
  apply ingred_lt. auto.
  assert (Hgt : size (E x y) > size (K y)).
  simpl. admit.      omega.
Qed.

Lemma CStrand_not_falling :
  ∀ (s:strand), CStrand s →
    ¬ ∃ (n1 n2 : node), recv n1 ∧ xmit n2 ∧
      strand_of n1 = s ∧ strand_of n2 = s ∧
      ingred (msg_of n2) (msg_of n1).
```

```
Proof.
intros s Hcs Hc.
destruct Hc as (n1,(n2,(Hre, (Hxmit,(Hs1,(Hs2,Hingred)))))).
inversion Hcs.
assert (Smn2 : smsg_of n2 = + P g h).
 apply strand_3_nodes_nnp_xmit with (x:=g) (y:=h).
 congruence. auto.
assert (Mn2 : msg_of n2 = P g h). unfold msg_of. rewrite Smn2. auto.
assert (Smn1 : smsg_of n1 = -g ∨ smsg_of n1 = -h).
  apply strand_3_nodes_nnp_recv with (x:=g) (y:=h) (z:=P g h).
  congruence. auto.
case Smn1.
  intro Sg. assert (Mn1 : msg_of n1 = g). unfold msg_of. rewrite Sg; auto.
  apply (pair_not_ingred_comp_l g h). rewrite Mn1, Mn2 in Hingred. auto.
  intro Sh. assert (Mn1 : msg_of n1 = h). unfold msg_of. rewrite Sh; auto.
  apply (pair_not_ingred_comp_r g h). rewrite Mn1, Mn2 in Hingred. auto.
Qed.

Lemma EStrand_not_falling :
  ∀ (s:strand), EStrand s →
    ¬ ∃ (n1 n2 : node), recv n1 ∧ xmit n2 ∧
      strand_of n1 = s ∧ strand_of n2 = s ∧
      ingred (msg_of n2) (msg_of n1).
Proof.
intros s Hes Hc.
destruct Hc as (n1,(n2,(Hre, (Hxmit,(Hs1,(Hs2,Hingred)))))).
inversion Hes.
assert (Smn2 : smsg_of n2 = + E h k).
 apply strand_3_nodes_nnp_xmit with (x:=K k) (y:=h).
 congruence. auto.
assert (Mn2 : msg_of n2 = E h k). unfold msg_of. rewrite Smn2. auto.
assert (Smn1 : smsg_of n1 = -K k ∨ smsg_of n1 = -h).
  apply strand_3_nodes_nnp_recv with (x:=K k) (y:=h) (z:=E h k).
  congruence. auto.
case Smn1.
  intro Sg.  assert (Mn1 : msg_of n1 = K k).  unfold msg_of.  rewrite Sg;
auto.
  apply (enc_not_ingred_comp_r h k). rewrite Mn1, Mn2 in Hingred. auto.
  intro Sh. assert (Mn1 : msg_of n1 = h). unfold msg_of. rewrite Sh; auto.
  apply (enc_not_ingred_comp_l h k). rewrite Mn1, Mn2 in Hingred. auto.
Qed.
```

5.5.1 A MStrand or KStrand cannot have an edge

```
Lemma strand_1_node_index_0 :
  ∀ x s, strand_of x = [s] → index_of x = 0.
Proof.
intros [[xs xn] xp] s Snx. simpl in *.
rewrite Snx in xp. simpl in xp. omega.
Qed.
```

```
Lemma MStrand_not_edge :
  ∀ (s:strand), MStrand s → ¬ ∃ (x y : node),
     strand_of x = s ∧ strand_of y = s ∧ ssuccs x y.
Proof.
intros s Ms (x ,(y, (Sx,(Sy, Sxy)))).
inversion Ms.
apply ssuccs_acyclic with (n:=x).
assert (Heq : x=y). apply eq_nodes.
congruence. assert (index_of x = 0).
apply strand_1_node_index_0 with (s := +T t). congruence.
assert (index_of y = 0).
apply strand_1_node_index_0 with (s := +T t). congruence.
congruence. congruence.
Qed.
```

```
Lemma KStrand_not_edge :
  ∀ (s:strand), KStrand s → ¬ ∃ (n1 n2 : node),
     strand_of n1 = s ∧ strand_of n2 = s ∧ ssuccs n1 n2.
Proof.
intros s Ms (x ,(y, (Sx,(Sy, Sxy)))).
inversion Ms.
apply ssuccs_acyclic with (n:=x).
assert (Heq : x=y). apply eq_nodes.
congruence. assert (index_of x = 0).
apply strand_1_node_index_0 with (s := +K k). congruence.
assert (index_of y = 0).
apply strand_1_node_index_0 with (s := +K k). congruence.
congruence. congruence.
Qed.
```

5.5.2 A CStrand or SStrand cannot have a transformed edge

```
Lemma CStrand_not_edge :
  ∀ (s:strand), CStrand s → ¬ ∃ (x y : node) (a :msg),
    strand_of x = s ∧ strand_of y = s ∧
    recv x ∧ xmit y ∧ transformed_edge x y a.
Admitted.
Axiom SStrand_not_edge :
∀ (s:strand), SStrand s → ¬ ∃ (x y : node) (a:msg),
    strand_of x = s ∧ strand_of y = s ∧
    recv x ∧ xmit y ∧ transformed_edge x y a.
```

5.6 Every inhabited predicate has a prec-minimal element

```
Theorem always_min_elt : ∀ P: node→ Prop,
  (∃ (x:node), (P x)) → has_min_elt P.
Proof.
  intros.
  destruct H.
  revert x H.
  unfold has_min_elt.
  unfold is_minimal.
  apply (@well_founded_ind node prec (wf_prec)
    (fun x:node ⇒
      P x → ∃ y:node, P y ∧ (∀ z:node, prec z y → ¬ (P z) ))).
  intros.
  case (classic (∀ y:node, P y → ¬ prec y x)).
  intros.
  ∃ x.
  split.
  auto.
  intros.
  assert (a2: P z → ¬ prec z x).
  apply H1.
  tauto.
  intros.
  apply not_all_ex_not in H1.
  destruct H1.
  apply imply_to_and in H1.
  destruct H1.
  apply NNPP in H2.
  apply H with (y:=x0).
  assumption.
  assumption.
Qed.
```

Prec is acyclic

```
Theorem prec_is_acyclic: ∀ (x:node), (prec x x) → False.
Proof.
  intros x H.
  assert (has_min_elt (fun x ⇒ prec x x )).
  apply always_min_elt.
  ∃ x; auto.
  unfold has_min_elt in H0.
  destruct H0.
  destruct H0.
  assert (prec x0 x0 → ¬ prec x0 x0 ).
  apply H1.
  tauto.
Qed.
```

5.7 Ingredients must originate

```
Section IngredientsOriginate.
  Variable the_m: msg.
  Definition m_ingred (n: node): Prop := ingred the_m (msg_of n).
```

If m is an ingredient somewhere then there is a minimal such place

```
  Lemma
    ingred_min:
    (∃ n:node, (m_ingred n)) →
    (∃ n:node, (is_minimal m_ingred n)).
  Proof.
    intros H.
    apply always_min_elt; auto.
  Qed.

Lemma smsg_xmit_msg :
  ∀ n m, smsg_of(n) = (+ m) → msg_of(n) = m.
Proof.
  intros n m H.
  unfold msg_of. rewrite H. reflexivity.
Qed.
```

```
Hint Resolve smsg_xmit_msg.

Lemma smsg_recv_msg :
  ∀ n m, smsg_of(n) = (- m) → msg_of(n) = m.
Proof.
  intros n m H.
  unfold msg_of. rewrite H. reflexivity.
Qed.

Hint Resolve smsg_recv_msg.

Lemma msg_deliver_msg_eq :
  ∀ x y, x --> y → msg_of x = msg_of y.
Proof.
  intros x y edge.
  destruct edge. destruct H as (H1, (H2, H3)).
  assert (msg_of(x) = m). apply smsg_xmit_msg. auto.
  assert (msg_of(y) = m). apply smsg_recv_msg. auto.
  congruence.
Qed.
```

A minimal node can't be a reception

```
  Lemma
    minimal_not_recv:
    ∀ (n:node), (is_minimal m_ingred n) →
      ¬ (recv n).
  Proof.
    unfold not.
    intros.
    unfold is_minimal in H.
    destruct H.
    assert (a1: ∃ n1 , msg_deliver n1 n).
    apply was_sent; auto.

    destruct a1.
    assert (a2: prec x n).
    apply deliver_prec; auto.

    assert (a3: ~(m_ingred x)).
    apply H1; auto.
    assert (a4: m_ingred x).

    assert (a5: msg_of x = msg_of n).
    apply msg_deliver_msg_eq. auto.
    unfold m_ingred.
    unfold m_ingred in H.
    rewrite a5; auto.
    tauto.
  Qed.
```

So, a minimal node must be a transmission

```
Lemma minimal_is_xmit: ∀ (n:node), (is_minimal m_ingred n) →
  (xmit n).
Proof.
  intros.
  case (xmit_or_recv n). auto.
  intro. apply False_ind. apply (minimal_not_recv n); auto.
Qed.
```

Main result of this section: an ingredient must originate

```
Theorem
  ingred_originates_2:
  (∃ n:node, (ingred the_m (msg_of n))) →
  (∃ n:node, (orig_at n the_m)).
Proof.
  intros.
  assert (a1: has_min_elt m_ingred).
  apply always_min_elt.
  unfold m_ingred; auto.
  unfold orig_at.
  unfold has_min_elt in a1.
  destruct a1.
  assert (a0: xmit x).
  apply minimal_is_xmit; auto.
  unfold m_ingred in H0.
  unfold is_minimal in H0.
  destruct H0.
  ∃ x.
  split.
  auto.
  split.
  auto.
  intros.
  assert (a1: prec n' x).
  apply ssuccs_prec; auto.
  revert H3.
  unfold not in H1.
  apply H1; auto.
Qed.
```

```
End IngredientsOriginate.
```

5.8 Extending two paths

```
Lemma path_nth_app_left :
  ∀ p q n, n < length p → nth_node n (p++q) = nth_node n p.
Proof.
intros p q n. apply list_nth_app_left.
Qed.

Lemma path_nth_app_right :
  ∀ p q n, n ≥ length p → n < length (p++q) →
    nth_node n (p++q) = nth_node (n-length p) q.
Proof.
intros p q n. apply list_nth_app_right.
Qed.

Lemma length_zero_nil : ∀ (p : list node), length p = 0 → p = [].
Proof.
intros. induction p. auto. simpl in H. omega.
Qed.

Lemma path_extend :
  ∀ (p : list node) (n:node) , is_path p →
    path_edge (nth_node (length p - 1) p) n → is_path (p++[n]).
Proof.
intros p n Pp Pe.
unfold is_path in *. intros i Hlt.
rewrite app_length in Hlt. simpl in *.
assert ( i < length p - 1 ∨ i = length p - 1).
omega. case H.
intros. repeat rewrite path_nth_app_left. apply Pp. auto. omega. omega.
intros.
  assert (length p = 0 ∨ length p > 0). omega.
  case H1. intros. rewrite (length_zero_nil p). rewrite app_nil_l. assert (i=0).
  omega. rewrite H3. simpl. omega. auto.
  intros. assert (i+1=length p). omega.
  rewrite path_nth_app_left. rewrite path_nth_app_right. rewrite H3.
  rewrite H0. rewrite minus_diag. apply Pe. omega. rewrite app_length.
  simpl. omega. omega.
Qed.

Lemma comp_of_node_imp_ingred :
  ∀ (m:msg) (n:node), m <[node] n → m <st (msg_of n).
```

```
Proof.
  intros.
  unfold comp_of_node in H.
  apply comp_imp_ingred.
  assumption.
Qed.
```

5.9 Transformation path

```
Section Trans_path.
Variable p : path.
Variable n : node.
Variable a t : msg.
Let lns := fst (split p).
Let lms := snd (split p).
Let n':= nth_node (length p - 1) lns.
Let t' := nth_msg (length p - 1) lms.
Lemma transpath_extend :
  is_trans_path p a → (path_edge n' n) ∨ (ssuccs n' n ∧ xmit n' ∧ xmit n) →
  (t' <[node] n' ∧ (t' = t ∨ (t'≠t → transformed_edge n' n a))) →
  a <st t → a <st t' →
  ((is_trans_path [(n',t'); (n,t)] a ∧ orig_at n' a) ∨ is_trans_path (p++[(n,t)])
a).
Proof.
  intros Tp Hor (Ct'n', C) At At'.
  destruct Tp.
  case Hor.
    intro Pe. right.
      split. case H.
      intro. left. unfold ln. rewrite list_split_fst.
      apply path_extend. auto. rewrite split_length_l. auto.

      intros. destruct H1 as (H2, (H3, (H4, H5))). right. unfold nd.
      unfold ln. repeat rewrite list_split_fst.
      repeat rewrite path_nth_app_left. split; auto.
      split. auto. split. auto. rewrite list_tl_extend. apply path_extend.
      auto.
Admitted.
End Trans_path.

Lemma comp_trans : ∀ a L n, a <st L → L <[node] n → a <st (msg_of n).
Proof.
intros a L n aL Ln.
```

```
apply ingred_trans with (y:= L).
auto. apply comp_of_node_imp_ingred. auto.
Qed.
Hint Resolve comp_trans.
Section Prop_11.
  Variable a L : msg.
  Variable n : node.
  Definition P_ingred : node → Prop:=
    fun (n':node) ⇒ ssuccs n' n ∧ ingred a (msg_of n').
  Definition P_comp : node → Prop :=
    fun (n':node) ⇒ ssuccs n' n ∧ L <[node] n' ∧ a <st L.
  Lemma P_comp_imp_P_ingred :
    ∀ x, P_comp x → P_ingred x.
  Proof.
    intros x Pcom.
    split. apply Pcom.
    apply comp_trans with (L:=L); apply Pcom.
  Qed.
  Lemma ingred_of_earlier :
        a <st (msg_of n) → xmit n → ¬ orig_at n a → ∃ n', P_ingred n'.
  Proof.
    intros Hst Hxmit Hnorig.
    apply Peirce.
    intros.
    apply False_ind.
    apply Hnorig. unfold orig_at.
    repeat split.
    auto.
    auto.
    intros n1 Hssuc Hastn1. apply H.
    ∃ n1. split; auto.
  Qed.
  Lemma new_at_earlier :
       a <st L → L <[node] n → ¬ new_at L n → ∃ n', P_comp n'.
  Proof.
    intros aL Can Nan. apply Peirce. intros.
    apply False_ind. apply Nan.
    split; auto. intros. apply H.
    ∃ n'. split. auto.
    split; auto.
  Qed.
```

```
Lemma not_orig_exists :
  a <st (msg_of n) → xmit n → ¬ orig_at n a → has_min_elt P_ingred.
Proof.
  intros Hxmit Hst Hnorig.
  apply always_min_elt.
  apply ingred_of_earlier; assumption.
Qed.
Hint Resolve not_orig_exists.

Lemma not_new_at_exists :
  a <st L → L <[node] n → ¬new_at L n → has_min_elt P_comp.
Proof.
  intros aL Can Nan. apply always_min_elt.
  apply new_at_earlier; auto.
Qed.

Lemma min_xmit_orig :
  ∀ (x:node), xmit x → is_minimal P_ingred x → orig_at x a.
Proof.
  intros. unfold orig_at. destruct H0. unfold P_ingred in H0.
  repeat split. auto. apply H0.
  intros. apply (H1 n'). apply ssuccs_prec. auto.
  unfold P_ingred. split. apply ssuccs_trans with (y:=x).
  auto. apply H0. auto.
 Qed.

Lemma min_new_at :
  ∀ (x:node), is_minimal P_comp x → new_at L x.
Proof.
  intros. destruct H.
  split. apply H.
  intros. apply (H0 n'). apply ssuccs_prec. auto.
  split. apply ssuccs_trans with (y:=x).
  auto. apply H. split; auto. apply H.
Qed.

Lemma eq_strand_trans :
  ∀ x y z, strand_of x = strand_of y → strand_of y = strand_of z →
  strand_of x = strand_of z.
Proof.
  intros x y z Sxy Syz.
  congruence.
Qed.

Lemma not_ssuccseq :
  ∀ (x y : node), ~(x ==>* y) → strand_of x = strand_of y → y ==>+ x.
  Proof.
    intros x y N Sxy. apply index_lt_ssuccs. auto.
    case lt_dec with (m:= index_of x) (n:= index_of y).
    intro. omega.
    intro. apply False_ind. apply N. apply index_lteq_ssuccseq.
    auto. omega.
  Qed.

  Lemma orig_precede_new_at :
    ∀ x y, is_minimal P_ingred x → is_minimal P_comp y → ssuccseq x y.
  Proof.
    intros x y Pin Pcom.
    apply Peirce. intros. assert (Syx : y ==>+ x).
    apply not_ssuccseq. auto.
    apply eq_strand_trans with (y:=n).
    apply ssuccs_same_strand. apply Pin.
    symmetry. apply ssuccs_same_strand. apply Pcom.
    apply False_ind. destruct Pin. apply (H1 y).
    apply ssuccs_prec; auto. apply P_comp_imp_P_ingred.
    apply Pcom.
  Qed.

End Prop_11.

Lemma msg_deliver_same_comp :
  ∀ x y Ly, msg_deliver x y → Ly <[node] y →
  ∃ Lx , Lx <[node] x ∧ Lx = Ly.
Proof.
  intros x y Ly Mxy Lyy.
  ∃ Ly. split. unfold comp_of_node.
  assert (msg_of x = msg_of y).
  apply msg_deliver_msg_eq. auto.
  rewrite H. auto. auto.
Qed.
```

For every atomic ingredient of a message, there exists a component of the message so that the atomic value is an ingredient of that component `Lemma ingred_exists_comp`:

```
  ∀ m a, atomic a → a <st m → ∃ L, a <st L ∧ comp L m.
Proof.
  intros m a Hatom Hingred.
  induction m.
  ∃ a; split.
  constructor.
  assert (a = T t). apply atomic_ingred_eq; auto.
  subst. apply comp_atomic_cyclic; assumption.

  ∃ a; split.
  constructor.
  assert (a = K k). apply atomic_ingred_eq; auto.
  subst. apply comp_atomic_cyclic; assumption.

  assert (Hor : ingred a m1 ∨ ingred a m2).
  apply ingred_pair. inversion Hatom; discriminate.
  assumption.
  case Hor.
  intro Hst.
  assert (Hex : ∃ L : msg, ingred a L ∧ comp L m1).
  exact (IHm1 Hst). destruct Hex as (L, (HaL, Hcom)).
  ∃ L; split.
  assumption.
  apply preserve_comp_l; assumption.

  intros Hst.
  assert (Hex : ∃ L : msg, ingred a L ∧ comp L m2).
  exact (IHm2 Hst). destruct Hex as (L, (HaL, Hcom)).
  ∃ L; split.
  assumption.
  apply preserve_comp_r; assumption.

  assert (Hex : ∃ L : msg, a <st L ∧ L <com m).
  apply IHm. apply ingred_enc with (k:=k).
  inversion Hatom; discriminate.
  assumption.
  destruct Hex as (L, (HaL, Hcom)).
  ∃ (E m k); split.
  assumption.
  apply comp_simple_cyclic.
  apply simple_step. unfold not. intros Hpair.
  inversion Hpair.
Qed.

Lemma ingred_exists_comp_of_node:
  ∀ (n:node) (a:msg), atomic a → a <st (msg_of n)
    → ∃ L, a <st L ∧ L <[node] n.
Proof.
  intros.
  apply ingred_exists_comp; assumption.
Qed.

Lemma msg_deliver_comp :
  ∀ (n1 n2:node) (m:msg),
    msg_deliver n1 n2 ∧
    comp_of_node m n2 → comp_of_node m n1.
Proof.
  intros.
  destruct H as (H1,H2).
  unfold comp_of_node.
  assert (msg_of n1 = msg_of n2).
  apply msg_deliver_msg_eq. auto.
  rewrite H.
  unfold comp_of_node in H2. auto.
Qed.
Hint Resolve msg_deliver_comp.

Lemma new_at_imp_comp : ∀ m n, new_at m n → m <[node] n.
Proof.
  intros m n H.
  unfold new_at in H.
  apply H.
Qed.

Lemma orig_dec : ∀ n a, orig_at n a ∨ ¬ orig_at n a.
Proof. intros; tauto. Qed.

Lemma new_at_dec : ∀ (n:node) (L:msg) , new_at L n ∨ ¬new_at L n.
Proof.
  intros n L. tauto.
Qed.

Lemma orig_imp_ingred : ∀ n a, orig_at n a → a <st msg_of n.
Proof.
intros. apply H.
Qed.

Lemma orig_precede :
  ∀ (x y : node) (a Ly : msg), atomic a → orig_at x a →
  a <st Ly → Ly <[node] y → strand_of x = strand_of y → ssuccseq x y.
Proof.
intros x y a Ly Atom Oxa aLy Lyy Sxy.
apply Peirce. intros. assert (Syx : y ==>+ x).
apply not_ssuccseq; auto. apply False_ind.
destruct Oxa as (_, (_, Contra)).
apply Contra with (n':=y). auto.
apply comp_trans with (L:=Ly); auto.
Qed.

Lemma ssuccseq_imp_eq_or_ssuccs :
  ∀ x y, ssuccseq x y → x = y ∨ ssuccs x y.
```

```
Proof.
intros x y Sxy. induction Sxy.
right. apply t_step; auto.
left. reflexivity.
case IHSxy1.
  intro. subst. auto.
  intro. case IHSxy2.
    intro. subst. auto.
    intro. right.
    apply ssuccs_trans with (y:=y); auto.
Qed.
```

5.10 Backward Constructions

```
Section back_ward.
  Variable a L: msg.
  Variable n : node.

Lemma backward_construction :
    atomic a → a <st L → L <[node] n → ¬ orig_at n a →
    ∃ (n':node) (L':msg), (path_edge n' n ∨ (ssuccs n' n ∧ xmit n' ∧ xmit n ∧
orig_at n' a)) ∧
      (a <st L' ∧ L' <[node] n' ∧ (L' = L ∨ (L'≠L → transformed_edge n' n
a))).
Proof.
  intros Hatom Hcom Hst Norig.
  case (xmit_or_recv n).
  Focus 2. intros Hrecv.
  assert (Hex : ∃ (n':node), msg_deliver n' n).
  apply was_sent. auto.
  destruct Hex as (n', Hmsg_deli).
  ∃ n'; ∃ L.
  split. left. apply path_edge_single. auto.
  split. exact Hcom.
  split. apply msg_deliver_comp with (n1:=n') (n2:=n).
  split; assumption.
  left. trivial.

  intros Hxmit.
    assert (Hmin : has_min_elt (P_ingred a n)).
    apply not_orig_exists. apply comp_trans with (L:= L); auto. auto. auto.
    destruct Hmin as (n', Hm).
    case (xmit_or_recv n').
```

```
      intros Xn'.
      assert (Orign : orig_at n' a).
      apply min_xmit_orig with (n:=n); auto.
      assert (Cn' : ∃ L', a <st L' ∧ L' <[node] n').
      apply ingred_exists_comp_of_node. auto.
      apply orig_imp_ingred; auto. destruct Cn' as (L', (aL', L'n')).
      case (new_at_dec n L).
        intros NLn.
        ∃ n', L'. split. right. split. destruct Hm. apply H. auto.
        split. auto. split. auto. case (eq_msg_dec L' L). auto.
        intros. right. intros. split. apply Hm. split; auto. ∃ n, L.
        split. apply Hm. split. apply rt_refl. split; auto.
        intros NNLn.
        assert (Hmin : has_min_elt (P_comp a L n)).
        apply not_new_at_exists; auto.
        destruct Hmin as (z, (H1, H2)).
        assert (Sn'z : ssuccseq n' z).
        apply orig_precede with (a:=a) (Ly:=L); auto.
        apply H1. apply eq_strand_trans with (y:=n).
        apply ssuccs_same_strand; auto. apply Hm. symmetry.
        apply ssuccs_same_strand. apply H1.
        case (ssuccseq_imp_eq_or_ssuccs n' z); auto.
          intros Eqn'z.
          ∃ n', L.
          split. right. split. apply Hm. split; auto.
          split; auto. split. subst. apply H1. left. auto.
          intros SSn'z.
          ∃ n', L'. split. right. split; auto. apply Hm.
          split; auto. split. auto.
          right. intros Neq. split. apply Hm. split; auto.
          ∃ z, L. split; auto. split. apply ssuccs_imp_ssuccseq. apply H1.
          split.  apply min_new_at with (a:=a) (n:=n).  split; auto.  split;
auto.
    intros Hrecv.
      assert (Cn' : ∃ L', a <st L' ∧ L' <[node] n').
      apply ingred_exists_comp_of_node. auto.
      apply Hm. destruct Cn' as (L', (aL', L'n')).
      case (new_at_dec n L).
      intros NLn.
      ∃ n', L'. split. left. apply path_edge_double. split; auto. apply Hm.
      split; auto. split. auto. right.
      intros Neq. split. apply Hm. split; auto. ∃ n, L.
      split. apply Hm. split. apply rt_refl. split. auto. split; auto.
    intros NNLn.
        assert (Hmin : has_min_elt (P_comp a L n)).
        apply not_new_at_exists; auto.
        destruct Hmin as (z, (H1, H2)).
        assert (Sn'z : ssuccseq n' z).
        apply orig_precede_new_at with (a:=a) (L:=L) (n:=n). auto.
        split; auto.
        case (ssuccseq_imp_eq_or_ssuccs n' z); auto.
          intros Eqn'z.
          ∃ n', L.
          split. left. apply path_edge_double; auto.
          split. apply Hm. split; auto.
          split; auto. split. subst. apply H1. left. auto.
          intros SSn'z.
          ∃ n', L'. split. left. apply path_edge_double; auto.
          split; auto. apply Hm. split; auto.
          split. auto.
          right. intros Neq. split. apply Hm. split; auto.
          ∃ z, L. split; auto. split. apply ssuccs_imp_ssuccseq. apply H1.
          split.
          apply min_new_at with (a:=a) (n:=n). split; auto. split; auto.
Qed.
End back_ward.
```

5.11 Others

```
Definition not_proper_subterm (t:msg) :=
  ∃ (n': node) (L : msg),
  t <st L → t ≠ L → r_node n' → L <[node] n' → False.
Definition r_comp (L:msg) (n:node) := L <[node] n ∧ r_node n.
Definition not_constant_tp (p:path) :=
  (nth_msg 0 (lm p)) ≠ (nth_msg (length p - 1) (lm p)).
Definition largest_index (p:path) (i:nat) :=
  not_constant_tp p ∧ i < length p - 1 ∧
  nth_msg i (lm p) ≠ nth_msg (i+1) (lm p) ∧
  ∀ j, j < length p → j > i →
  nth_msg j (lm p) = nth_msg (length p - 1) (lm p).
```

```
Definition smallest_index (p:path) (i:nat) :=
  not_constant_tp p ∧ i < length p - 1 ∧
  nth_msg i (lm p) ≠ nth_msg (i+1) (lm p) ∧
  ∀ j, j ≤ i → nth_msg j (lm p) = nth_msg 0 (lm p).

Lemma largest_index_imp_eq_last :
  ∀ p i j, largest_index p i → j < length p → j > i →
  nth_msg j (lm p) = nth_msg (length p - 1) (lm p).
Proof.
intros.
apply H; auto.
Qed.

Lemma not_constant_exists :
  ∀ p, not_constant_tp p → ∃ i , i < length p - 1 →
  nth_msg i (lm p) ≠ nth_msg (i+1) (lm p).
Admitted.

Lemma not_constant_exists_smallest :
  ∀ p, not_constant_tp p → ∃ i , smallest_index p i.
Admitted.

Lemma not_constant_exists_largest :
  ∀ p, not_constant_tp p → ∃ i , largest_index p i.
Admitted.

Lemma strand_length_3 :
  ∀ (s:strand) (x y z : smsg), s = [x ; y ; z] → length s = 3.
Proof.
intros. unfold length. subst. auto.
Qed.

Lemma DS_exists_key :
  ∀ y h k k', DStrand (strand_of y) → msg_of y = E h k → inv k k' →
  ∃ x , ssuccs x y ∧ msg_of x = K k'.
Admitted.

Lemma DS_node_0 :
  ∀ x, DStrand (strand_of x) → index_of x = 0 → ∃ k , msg_of x = K k.
Admitted.

Lemma DS_node_1 :
  ∀ x, DStrand (strand_of x) → (∃ h k , msg_of x = E h k) → index_of x = 1.
Admitted.

Lemma msg_of_nth :
  ∀ p n, n < length p → msg_of (nd p n) = nth_msg n (lm p).
Admitted.
```

Chapter 6

Authentication_Tests_Library

This chapter contains the proofs of all propositions needed for authentication tets.

```
Require Import Strand_Spaces Message_Algebra Strand_Library.
Require Import Lists.List Relation_Definitions Relation_Operators.
Require Import List_Library.
Import ListNotations.
```

6.1 Proposition 6

A destructive path that enters decryption strands only through D-cyphertext edges is falling [6].

```
Lemma P6_1 : ∀ p, des_path_not_key p → falling_path p.
Proof.
  intros p Dp. destruct Dp.
  split. apply H.
  intros i Hlt. destruct H as (pp, H). destruct pp.
  unfold is_path in H1.
  assert (path_edge (nth_node i p) (nth_node (i + 1) p)).
  apply (H1 i); auto. inversion H3.
  assert (msg_of (nth_node i p) = msg_of (nth_node (i+1) p)).
  apply msg_deliver_msg_eq with (y:=nth_node (i+1) p). auto.
  rewrite H5. apply ingred_refl. destruct H4 as (H5, (H6, H7)).
Admitted.
```

A constructive path that enters encryption strands only through E-plaintext edges is rising [6]

```
Lemma P6_2 : ∀ p, cons_path_not_key p → rising_path p.
```

Admitted.

6.2 Proposition 7

The sequence of penetrator strands traversed on a falling path is constrained by the structure of term(p1).

```
Section P7_1.
  Variable i : nat.
  Variable p : list node.
  Let p_i := nth_node i p.
  Let p_i1 := nth_node (i+1) p.
  Hypothesis Hc : 0 < i ∧ i < length p - 1.
  Hypothesis Hfp : falling_path p.
  Hypothesis Hrec : recv p_i.
  Hypothesis Hpn : p_node p_i.
  Let s := strand_of p_i.

  Lemma path_edge_pi_pi1 : path_edge p_i p_i1.
  Proof.
  destruct Hfp as (Hp,_). destruct Hp as (P, _).
  unfold is_path in P.
  apply (P i); apply Hc.
  Qed.

  Lemma P7_1_aux1 : xmit p_i1 ∧ strand_of p_i1 = strand_of p_i.
  Proof.
  assert (Pe : path_edge p_i p_i1).
  apply path_edge_pi_pi1; auto.
  inversion Pe. apply False_ind. apply xmit_vs_recv with (n:=p_i).
  apply msg_deliver_xmit with (y:=p_i1). auto. auto.
  split. apply H.
  destruct H. symmetry. apply ssuccs_same_strand. auto.
  Qed.

  Lemma pi1_ingred_pi : msg_of p_i1 <st msg_of p_i.
  Proof.
  destruct Hfp. apply (H0 i). apply Hc.
  Qed.

  Lemma P7_1_aux : DStrand (strand_of p_i) ∨ SStrand (strand_of p_i).
  Proof.
    assert (Ps : PenetratorStrand s). apply Hpn.
    inversion Ps.
    apply False_ind. apply xmit_vs_recv with (n:=p_i).
      apply MStrand_xmit_node; auto. auto.

    apply False_ind. apply xmit_vs_recv with (n:=p_i).
      apply KStrand_xmit_node; auto. auto.

    apply False_ind. apply (CStrand_not_falling s) ; auto.
      ∃ p_i, p_i1. split; auto. split. apply P7_1_aux1.
      split; auto. split. apply P7_1_aux1. apply pi1_ingred_pi.

    auto.

    apply False_ind. apply False_ind. apply (EStrand_not_falling s) ; auto.
      ∃ p_i, p_i1. split; auto. split. apply P7_1_aux1.
      split; auto. split. apply P7_1_aux1. apply pi1_ingred_pi.

    auto.
  Qed.

Section P7_1_a.
  Variable h : msg.
  Variable k : Key.
  Hypothesis Heq : msg_of p_i = E h k.
  Lemma P7_1a :
    DStrand s ∧ msg_of p_i1 = h.
  Proof.
    case P7_1_aux.
    intro Ds. split; auto.
    inversion Ds.
    assert (Smpi : smsg_of p_i = - K k' ∨ smsg_of p_i = - E h0 k0).
      apply strand_3_nodes_nnp_recv with (z:=h0); auto.
      case Smpi. intro Kk. assert (msg_of p_i = K k').
      unfold msg_of. rewrite Kk; auto. rewrite H1 in Heq. discriminate.
   intro. assert (msg_of p_i = E h0 k0). unfold msg_of; rewrite H1; auto.
     rewrite Heq in H2. assert (h=h0 ∧ k = k0). apply ((enc_free h k h0 k0));
auto.
     destruct H3; subst.
     apply node_smsg_msg_xmit.
     apply strand_3_nodes_nnp_xmit with (x:= K k') (y:=E h0 k0).
     assert (strand_of p_i1 = strand_of p_i). apply P7_1_aux1. congruence.
     apply P7_1_aux1.
  intro. inversion H.
    assert (smsg_of p_i = - P g h0).
    apply strand_3_nodes_npp_recv with (y:= g) (z:=h0). auto. auto.
    assert (msg_of p_i= P g h0). apply node_smsg_msg_recv; auto.
    rewrite H2 in Heq. discriminate.
  Qed.
```

```
End P7_1_a.

Section P7_1_b.
  Variable g h : msg.
  Lemma P7_1_b :
    SStrand s ∧ (msg_of p_i1 = h ∨ msg_of p_i1 = g).
Admitted.
End P7_1_b.
End P7_1.
```

6.3 Proposition 10

This lemma states that if (p, L) is a transformation path in which $L_i \neq L_{i+1}$, and p_i is a penetrator node, then $p_i \Rightarrow^+ p_{i+1}$ lies either on a D-strand or an E-strand [6].

```
Section Proposition_10.
  Variable p : path.
  Variable n : nat.
  Variable a : msg.
  Hypothesis Htp : is_trans_path p a.
  Hypothesis Hn : n < length p - 1.
  Hypothesis Hcom : L p n ≠ L p (n+1).
  Hypothesis Pnode : p_node (nd p n).

  Lemma trans_path_ssuccs :
      ssuccs (nd p n) (nd p (n+1)).
  Proof.
    destruct Htp as (H2, (H3,H4)).
    destruct (H4 n)as (H41, H42).
    destruct H42. auto.
      apply False_ind. apply Hcom. auto.
    destruct H. auto.
    destruct H0 as (m,(Hxmit,(Hnew,(Hssuc,Hsseq)))).
    auto.
  Qed.

  Lemma Prop10_recv_xmit : recv (nd p n) ∧ xmit (nd p (n+1)).
  Admitted.

  Lemma Proposition_10 : ssuccs (nd p n) (nd p (n+1)) ∧
    (DStrand (strand_of (nd p n)) ∨ EStrand (strand_of (nd p n))).
  Proof.
    destruct Htp as (Ha, (Ha', Hb)).
    destruct (Hb n) as (Q1,Q2).
    split.
      apply trans_path_ssuccs.
    assert (Hp : PenetratorStrand (strand_of (nd p n))).
    apply (P_node_strand (nd p n)); auto.
    elim Hp.
    intro. apply False_ind. apply (MStrand_not_edge (strand_of (nd p n))).
    auto.
    ∃ (nd p n).
    ∃ (nd p (n+1)).
    split. auto.
    split. symmetry. apply ssuccs_same_strand. apply trans_path_ssuccs.
    apply trans_path_ssuccs.
    intro. apply False_ind. apply (KStrand_not_edge (strand_of (nd p n))).
    auto.
    ∃ (nd p n).
    ∃ (nd p (n+1)).
    split. auto.
    split. symmetry. apply ssuccs_same_strand; apply trans_path_ssuccs.
    apply trans_path_ssuccs.
    intro. apply False_ind. apply (CStrand_not_edge (strand_of (nd p n))).
    auto.
    ∃ (nd p n),(nd p (n+1)), a.
    split. auto.
    split. symmetry. apply ssuccs_same_strand. apply trans_path_ssuccs.
    split. apply Prop10_recv_xmit.
    split. apply Prop10_recv_xmit.
    case (Q2 Hn). intro. apply False_ind. apply Hcom. auto.
    intro. apply (H0 Hcom).
    intro. apply False_ind. apply (SStrand_not_edge (strand_of (nd p n))).
    auto.
    ∃ (nd p n),(nd p (n+1)), a.
    split. auto.
    split. symmetry. apply ssuccs_same_strand. apply trans_path_ssuccs.
    split. apply Prop10_recv_xmit.
    split. apply Prop10_recv_xmit.
    case (Q2 Hn). intro. apply False_ind. apply Hcom. auto.
    intro. apply (H0 Hcom).
    auto.
    auto.
  Qed.
End Proposition_10.
```

6.4 Proposition 11

This propositions states that given a node such that an atomic message a is an ingredient of the node's message, it is possible to construct a transformation path so that the atomic value is originated at the first node of the path and the given node is the last node of the path.

```
Section Proposition_11.

Lemma single_node_tp :
  ∀ (n:node) (m a:msg),
    atomic a → a <st m → m <[node] n → is_trans_path [(n,m)] a.
Proof.
  intros n m a Atom Ingred Hcom.
  unfold is_trans_path.
  simpl. split. left. unfold is_path. simpl. intros i Hcontra.
  apply False_ind; omega.

  split. auto.
  intros n0. split. intro Hn0. assert (n0=0). omega. rewrite H.
  assert ( L [(n,m)] 0 = m). auto.
  assert (nd [(n,m)] 0 = n). auto.
  split; congruence.

  intros n1.
  apply False_ind. omega.
Qed.

Lemma single_node_not_traverse_key :
  ∀ (n:node) (m a : msg), atomic a → a <st m → m <[node] n →
  is_trans_path [(n,m)] a → not_traverse_key [(n,m)].
Proof.
intros.
unfold not_traverse_key. intros.
simpl in H3. omega.
Qed.

Definition p11_aux (n:node) (a t : msg) p : Prop :=
  let ln := fst (split p) in
  let lm := snd (split p) in
  is_trans_path p a ∧
  orig_at (nth_node 0 ln) a ∧
  nth_node (length p - 1) ln = n ∧
  nth_msg (length p -1) lm = t ∧
  ∀ (i:nat), i < length p → a <st (nth_msg i lm) ∧
  not_traverse_key p.
```

```
Definition p11_aux2 (n:node): Prop :=
  ∀ (a t : msg), atomic a → a <st t → t <[node] n →
  ∃ p, p11_aux n a t p.

Lemma tpath_extend :
  ∀ x a t, a <st t → t <[node] x →
  (∃ (x':node) (t':msg), (path_edge x' x ∨ (ssuccs x' x ∧ xmit x' ∧ xmit x ∧
orig_at x' a)) ∧
  (a <st t' ∧ t' <[node] x' ∧ (t' = t ∨ (t'≠t → transformed_edge x' x a))) ∧
  ∃ p, p11_aux x' a t' p) →
  ∃ p, p11_aux x a t p.
Proof.
Admitted.

Lemma Prop_11 : ∀ (n' : node) , p11_aux2 n'.
Proof.
apply well_founded_ind with (R:=prec).
exact wf_prec.
  intros x IH.
  intros a t Sat Atoma Stx.
  assert (Orig : orig_at x a ∨ ¬ orig_at x a). tauto.
  case Orig.
  intros Oxa. ∃ ([(x, t)]). split.
  apply single_node_tp with (n:=x) (m:=t); auto.
  split; auto. split; auto. split; auto.
  intros. simpl in H. assert (i=0). omega.
  split.  rewrite H0;auto.  apply single_node_not_traverse_key with (a:=a);
auto.
  apply single_node_tp with (n:=x) (m:=t); auto.

  intro NOrig. case (xmit_or_recv x).
  Focus 2. intro Recvx. assert (∃ y, msg_deliver y x).
  apply was_sent; auto. apply tpath_extend; auto. destruct H as (y, Dyx).
  ∃ y, t. split. left. apply path_edge_single. auto.
  split. split. auto. split. apply msg_deliver_comp with (n2:=x).
  split; auto. left; auto. apply IH. apply deliver_prec; auto. auto.
  auto. apply msg_deliver_comp with (n2:=x). split; auto.

  intros.
  assert (∃ (x':node) (t':msg),
          (path_edge x' x ∨ (ssuccs x' x ∧ xmit x' ∧ xmit x ∧ orig_at x' a)) ∧
          (a <st t' ∧ t' <[node] x' ∧ (t' = t ∨ (t'≠t → transformed_edge x' x
a)))).
  apply backward_construction; auto. destruct H0 as (y, (Ly, (H1, H2))).
  apply tpath_extend; auto. ∃ y, Ly. split. apply H1.
  split. apply H2.
  apply IH. case H1.
    intro. apply path_edge_prec. auto.
    intro. apply ssuccs_prec. apply H0.
  auto. apply H2. apply H2.
Qed.
End Proposition_11.
```

6.5 Proposition 13

```
Section P13.
  Variable pl : path.
  Let p := fst (split pl).
  Let l := snd (split pl).
  Hypothesis Hpp : p_path p.
  Hypothesis Hp1 : simple (msg_of (nth_node 0 p)).
  Lemma Prop13 :
    ∀ (i:nat), i < length p - 1 →
    ∃ (j:nat), (j ≤ i ∧ msg_of (nth_node j p) = nth_msg i l).
  Admitted.
  Definition P13_1_aux (n:nat) : Prop :=
    msg_of (nth_node n p) = (nth_msg (length p - 1) l) ∧
    ∀ (i:nat), i ≥ n → i ≤ length p - 1 →
      nth_msg i l = nth_msg (length p - 1) l.
  Lemma P13_1 :
    ∃ (n:nat), P13_1_aux n ∧
      (∀ m, m > n → ¬ P13_1_aux m) ∧
      ∃ i, i < length p - 1 → nth_msg i l ≠ nth_msg (i+1) l →
        xmit (nth_node n p) ∧ EStrand (strand_of (nth_node n p)).
  Admitted.
End P13.
```

6.6 Proposition 17

This lemma states that either a penetrable key is already penetrated, or some regular principal puts it in a form that could allow it to be penetrated. In fact, any key that becoms available to the penetrator in any bundle is a member of PKeys [6].

```
Section P17.
Definition Prop17_aux (n:node) : Prop :=
  ∀ (k : Key), msg_of n = K k → PKeys k.
Lemma Prop17 : ∀ (n:node), Prop17_aux n.
Proof.
  apply well_founded_ind with (R:=prec).
  exact wf_prec.
  intros x IH. unfold Prop17_aux in *.
Admitted.
End P17.
```

6.7 Proposition 18

```
Section P18.
Variable p : path.
Variable a : msg.
Hypothesis t_path: is_trans_path p a.
Hypothesis no_key : not_traverse_key p.
Hypothesis p1 : r_node (nth_node 0 (In p)).
Hypothesis lp : r_node (nth_node (length p - 1) (In p)).
Hypothesis nconst : (nth_msg 0 (Im p)) ≠ (nth_msg (length p - 1) (Im p)).
  Section P18_1.
  Variable h1 : msg.
  Variable k1 k1' : Key.
  Hypothesis enc_form : nth_msg 0 (Im p) = E h1 k1.
  Hypothesis key_pair : inv k1 k1'.
  Hypothesis not_pen : ¬PKeys k1'.
  Hypothesis not_subterm : not_proper_subterm (nth_msg 0 (Im p)).
  Lemma Prop18_1 :
    ∀ n, smallest_index p n →
    r_node (nth_node n (In p)) ∧
    transforming_edge_for (nth_node n (In p)) (nth_node (n+1) (In p)) a.
  Proof.
  intros n Sm.
  split. unfold r_node. intro pn.
  destruct Sm as (nc, (S1, (S2, S3))).
  assert (ssuccs (nd p n) (nd p (n+1)) ∧
            (DStrand (strand_of (nd p n)) ∨ EStrand (strand_of (nd p n)))).
  apply Proposition_10 with (a:=a); auto.
  destruct H.
  case H0.
  intro ds. apply not_pen.
  assert (nth_msg n (Im p) = E h1 k1). rewrite ← enc_form. apply S3. auto.
  assert (∃ x, ssuccs x (nd p n) ∧ msg_of x = K k1').
  apply DS_exists_key with (h:=h1) (k:=k1). auto.
  rewrite ← H1. apply msg_of_nth. omega. auto.
  destruct H2 as (x, (H2, H3)).
  assert (Prop17_aux x). apply Prop17. apply H4. unfold Prop17_aux in H1.
  auto.

  intros es.
  Admitted.
  End P18_1.

  Section P18_2.
  Variable hp : msg.
  Variable kp kp' : Key.
  Hypothesis enc_form : nth_msg (length p - 1) (Im p)= E hp kp.
  Hypothesis key_pair : inv kp kp'.
  Hypothesis not_pen : ¬PKeys kp'.
  Hypothesis not_subterm : not_proper_subterm (nth_msg (length p - 1) (Im p)).

  Lemma Prop18_2 :
    ∀ n, largest_index p n →
    r_node (nth_node n (In p)) ∧
    transforming_edge_for (nth_node n (In p)) (nth_node (n+1) (In p)) a.
  Admitted.
  End P18_2.
End P18.
```

Chapter 7

Authentication_Tests

This chapter contains the proofs of the two authentication tests, outgoing test and incoming test, which are the main results of this project.

```
Require Import Strand_Spaces Strand_Library Message_Algebra
               Authentication_Tests_Library.
```

7.1 Definitions

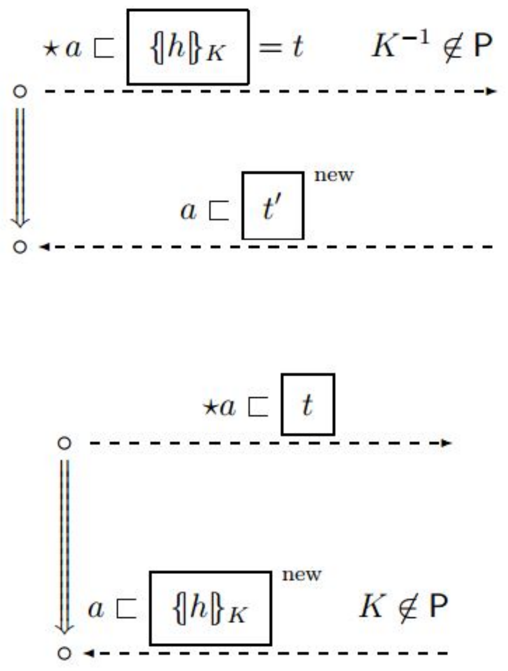
7.1.1 Test component and test

Tests can use their test components in at least two different ways. If the uniquely originating value is sent in encrypted form, and the challenge is to decrypt it, then that is an outgoing test. If it is received back in encrypted form, and the challenge is to produce that encrypted form, then that is an incoming test [6]. These two kinds of test are illustrated in Figure 7.1.

```
Definition test_component (a t: msg) (n:node) : Prop :=
  (∃ h k, t = E h k) ∧ a <st t ∧ t <[node] n ∧ not_proper_subterm t.
Definition test (x y : node) (a : msg) : Prop :=
  unique a ∧ orig_at x a ∧ transformed_edge_for x y a.
```

7.1.2 Incoming test

```
Definition incoming_test (x y : node) (a t: msg) : Prop :=
  (∃ h k, t = E h k ∧ ¬ PKeys k) ∧ test x y a ∧ test_component a t y.
```



$\star$ means a originates uniquely here

$\boxed{t}$ means t is a component of this node

Figure 7.1: Outgoing and Incoming Tests

Outgoing test `Definition` outgoing_test (x y : node) (a t : **msg**) : `Prop` :=
(∃ h k k', t = E h k ∧ *inv* k k' ∧ ¬ **PKeys** k') ∧
test x y a ∧ test_component a t x.

7.2 Some basic results

Below are some basic results following directly form the definitions for test, test component, outgoing test, and incoming test.

7.2.1 Unique

```
Lemma test_imp_unique : ∀ x y a, test x y a → unique a.
Proof.
intros. apply H.
Qed.
Hint Resolve test_imp_unique.
```

```
Lemma incoming_test_imp_unique :
  ∀ x y a t, incoming_test x y a t → unique a.
Proof.
intros. apply H.
Qed.
Hint Resolve incoming_test_imp_unique.
```

```
Lemma outgoing_test_imp_unique :
  ∀ x y a t, outgoing_test x y a t → unique a.
Proof.
intros. apply H.
Qed.
Hint Resolve outgoing_test_imp_unique.
```

7.2.2 Transformed edge

```
Lemma test_imp_trans_edge :
  ∀ x y a, test x y a → transformed_edge_for x y a.
Proof.
intros. apply H.
Qed.
Hint Resolve test_imp_trans_edge.
```

```
Lemma incoming_test_imp_trans_edge :
  ∀ x y a t, incoming_test x y a t → transformed_edge_for x y a.
```

Proof.
intros. apply H.
Qed.
Hint Resolve incoming_test_imp_trans_edge.

Lemma outgoing_test_imp_trans_edge :
  $\forall$ x y a t, outgoing_test x y a t $\rightarrow$ transformed_edge_for x y a.
Proof.
intros. apply H.
Qed.
Hint Resolve outgoing_test_imp_trans_edge.

Origination Lemma test_imp_orig : $\forall$ x y a, test x y a $\rightarrow$ orig_at x a.
Proof.
intros. apply H.
Qed.
Hint Resolve test_imp_orig.

Lemma incoming_test_imp_orig :
  $\forall$ x y a t, incoming_test x y a t $\rightarrow$ orig_at x a.
Proof.
intros. apply H.
Qed.
Hint Resolve incoming_test_imp_orig.

Lemma outgoing_test_imp_orig :
  $\forall$ x y a t, outgoing_test x y a t $\rightarrow$ orig_at x a.
Proof.
intros. apply H.
Qed.
Hint Resolve outgoing_test_imp_orig.

7.2.3 Ingredient

Lemma tc_ingred : $\forall$ a t n, test_component a t n $\rightarrow$ a <st t.
Proof.
intros a t n Tc.
apply Tc.
Qed.
Hint Resolve tc_ingred.

7.2.4 Incoming test (outging test) implies test_component

Lemma incoming_test_imp_tc :

$\forall$ x y a t, incoming_test x y a t $\rightarrow$ test_component a t y.
Proof.
intros. apply H.
Qed.
Hint Resolve incoming_test_imp_tc.

Lemma outgoing_test_imp_tc :
$\forall$ x y a t, outgoing_test x y a t $\rightarrow$ test_component a t x.
Proof.
intros. apply H.
Qed.
Hint Resolve outgoing_test_imp_tc.

Component Lemma tc_comp : $\forall$ a t n, test_component a t n $\rightarrow$ t <[node] n.
Proof.
intros a t n Tc.
apply Tc.
Qed.
Hint Resolve tc_comp.

Lemma outgoing_test_comp :
$\forall$ x y a t, outgoing_test x y a t $\rightarrow$ t <[node] x.
Proof.
intros. apply tc_comp with $(a{:=}a)$.
apply outgoing_test_imp_tc with $(y{:=}y)$.
auto.
Qed.
Hint Resolve outgoing_test_comp.

Lemma incoming_test_comp :
$\forall$ x y a t, incoming_test x y a t $\rightarrow$ t <[node] y.
Proof.
intros. apply tc_comp with $(a{:=}a)$.
apply incoming_test_imp_tc with $(x{:=}x)$.
auto.
Qed.
Hint Resolve incoming_test_comp.

Others Lemma unique_orig :
$\forall$ x y a, unique a $\rightarrow$ orig_at x a $\rightarrow$ orig_at y a $\rightarrow$ x = y.
Proof.
intros. destruct H. apply $(H2\ x\ y)$; auto.
Qed.

Lemma transpath_not_constant :
$\forall$ p a, is_trans_path p a $\rightarrow$

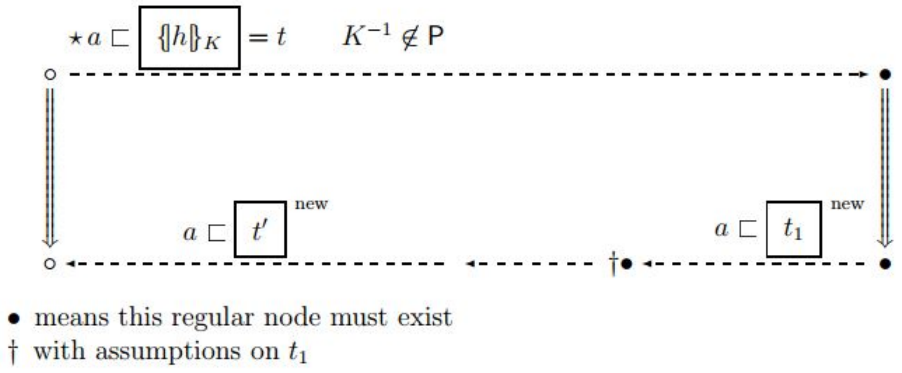
```
  transformed_edge_for (nth_node 0 (ln p)) (nth_node (length p - 1) (ln p)) a →
  not_constant_tp p.
Admitted.

Lemma ssuccs_both_r_nodes :
  ∀ x y, ssuccs x y → r_node x → r_node y.
Proof.
intros.
unfold r_node in *. unfold p_node in *.
rewrite (ssuccs_same_strand x y) in H0; auto.
Qed.

Lemma trans_ef_imp_ssuccs :
  ∀ x y a, transforming_edge_for x y a → ssuccs x y.
Proof.
intros. apply H.
Qed.
Hint Resolve trans_ef_imp_ssuccs.

Lemma tp_comp :
  ∀ p a i, is_trans_path p a → i < length p →
  nth_msg i (lm p) <[node] nth_node i (ln p).
Proof.
intros. apply H. auto.
Qed.

Lemma tf_edge_exists :
  ∀ x y a, transformed_edge_for x y a →
  ∃ Ly, a <st Ly ∧ Ly <[node] y.
Proof.
intros.
destruct H as ((H1, (H2, (z, (Ly, (H3, (H4, (H5, (H6, H7)))))))), (H8, H9)).
∃ Ly. auto.
Qed.
```

7.3 Aunthentication tests

```
Section Authentication_tests.
Variable n n' : node.
Variable a t: msg.
Hypothesis Atom : atomic a.
```

7.3.1 Outgoing test

If a regular pricipal sends out a messages in encrypted form, the original component, and sometime later receives it back in a new component. Then we can conclude that there exists a regular transforming edge. The meaning of this test is illusrated in the Figure 7.2.



$\star a \sqsubset \boxed{\{\!|h|\!\}_K} = t$
$K^{-1} \notin P$
$a \sqsubset \boxed{t'}$ new
$a \sqsubset \boxed{t_1}$ new
† ●
● means this regular node must exist
† with assumptions on t_1

Figure 7.2: Authentication provided by an Outgoing Test

```
Theorem Authentication_test1 :
  outgoing_test n n' a t →
  ∃ m m', r_node m ∧ r_node m' ∧ t <[node] m ∧
  transforming_edge_for m m' a.
Proof.
intros.
assert (p11_aux2 n').
apply Prop_11.
assert (Ha : ∃ t', a <st t' ∧ t' <[node] n').
apply tf_edge_exists with (x:=n).
apply outgoing_test_imp_trans_edge with (t:=t). auto.
destruct Ha as (t', (Hst, Hcomp)).
destruct (H0 a t'); auto.
destruct H1. destruct H2 as (H2, (H3, (H4, H5))).
assert (nth_node 0 (ln x) = n).
apply unique_orig with (a:=a).
apply outgoing_test_imp_unique with (x:=n) (y:=n') (t:=t). auto.
apply H2. apply outgoing_test_imp_orig with (y:=n') (t:=t). auto.
assert (not_constant_tp x).
apply transpath_not_constant with (a:=a). auto.
apply outgoing_test_imp_trans_edge with (t:=t).
unfold ln in *. rewrite H3. rewrite H6. auto.
assert (∃ i, smallest_index x i).
```

```
apply not_constant_exists_smallest. auto.
destruct H8 as (i, H8).
∃ (nth_node i (In x)), (nth_node (i+1) (In x)).
assert (r_node (nth_node i (In x)) ∧
transforming_edge_for (nth_node i (In x)) (nth_node (i + 1) (In x)) a).
apply Prop18_1. apply H8. destruct H8 as (H8, (H81, (H82, H83))).
split. apply H9.
split. apply ssuccs_both_r_nodes with (x := nth_node i (In x)).
apply trans_ef_imp_ssuccs with (a:=a); apply H9. apply H9.
split. assert (nth_msg i (snd (List.split x)) =
                  nth_msg 0 (snd (List.split x))).
apply H83. omega.
assert (nth_msg 0 (Im x) = t). admit. unfold Im in H11. rewrite H11 in H10.
unfold In. rewrite ← H10.
apply tp_comp with (a:=a). auto. omega. apply H9.
Qed.
```

7.3.2 Incoming test

Incoming tests can be used to infer the existence of a regular transforming edge in protocols in which the nonce is emitted in paintext, and later received in cnrypted form [6].

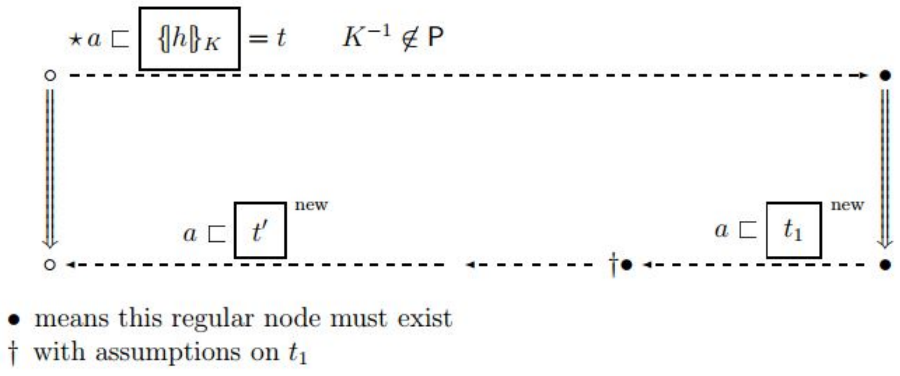

$\star a \sqsubset \boxed{\{\!|h|\!\}_K} = t$
$K^{-1} \notin P$
new
$a \sqsubset \boxed{t'}$
new
$a \sqsubset \boxed{t_1}$
†
● means this regular node must exist
† with assumptions on t_1

Figure 7.3: Authentication provided by an Incoming Test

```
Theorem Authentication_test2 :
  incoming_test n n' a t →
  ∃ m m', r_node m ∧ r_node m' ∧ t <[node] m' ∧
  transforming_edge_for m m' a.
Proof.
intros.
```

```
assert (p11_aux2 n').
apply Prop_11. destruct (H0 a t). auto.
apply tc_ingred with (n:=n').
apply incoming_test_imp_tc with (x:=n). auto.
apply incoming_test_comp with (x:=n) (a:=a). auto.
destruct H1. destruct H2 as (H2, (H3, (H4, H5))).
assert (nth_node 0 (In x) = n).
apply unique_orig with (a:=a).
apply incoming_test_imp_unique with (x:=n) (y:=n') (t:=t). auto.
apply H2. apply incoming_test_imp_orig with (y:=n') (t:=t). auto.
assert (not_constant_tp x).
apply transpath_not_constant with (a:=a). auto.
apply incoming_test_imp_trans_edge with (t:=t).
unfold In in *. rewrite H3. rewrite H6. auto.
assert (∃ i, largest_index x i).
apply not_constant_exists_largest. auto.
destruct H8 as (i, H8).
∃ (nth_node i (In x)), (nth_node (i+1) (In x)).
assert (r_node (nth_node i (In x)) ∧
transforming_edge_for (nth_node i (In x)) (nth_node (i + 1) (In x)) a).
apply Prop18_2. apply H8. destruct H8 as (H8, (H81, (H82, H83))).
split. apply H9.
split. apply ssuccs_both_r_nodes with (x := nth_node i (In x)).
apply trans_ef_imp_ssuccs with (a:=a); apply H9. apply H9.
split. assert (nth_msg (i+1) (snd (List.split x)) =
                  nth_msg (length x - 1) (snd (List.split x))).
apply H83. omega. omega.
rewrite H4 in H10. unfold In. rewrite ← H10.
apply tp_comp with (a:=a). auto. omega. apply H9.
Qed.

End Authentication_tests.
```

Chapter 8

Conclusion and Future Work

I successfully formalized strand spaces in Coq, and had the proofs for the two authentication tests with some incomplete proofs. To accomplish these, I implemented 6 modules as following.

1. Message Algebra: formalization of possible messages which can be exchanged between principals in a protocol.
2. Strand Spaces: formalization of node, strand, penetrator strand, edges, etc.
3. Strand Library: many basic results of strand spaces, which are used to prove the authentication tests
4. Authentication Library: the proofs of all propositions needed for proving authentication tests
5. List Library: some basic results about lists not found in the standard Coq List library
6. Authentication Tests: the proofs of the two main theorems

8.1 Future Work

This section describes the possible future work that can be done based on the project.

We already have a framework, strand space formalization, for specifying and verifying cryptographic protocols in general. One potential project following this one is to specify and verify some particular protocols like Otway-Rees, Woo-Lam, Newman-Stubblebine, then apply "Authentication Tests" to prove some specific security goals of these protocols.

When formalizing strand spaces in Coq, I encountered a lot of design choices and I had to decide which option to use, for example, inductive definitions (starting with "Fixpoint" in Coq) and deductive definitions (starting with "Definition" in Coq), variable and parameter. Each design choice has some advantages and some disadvantages. So the answer to which one is better depends on the usages of it later.

We can use Coq to extract programs from proofs. So we can use this Coq's facility to extract the programs of cryptographic protocols, and then use such programs to synthesize protocol implementations. It is a good way to detect the protocol failures or protocol errors.

Due to the limit of time and the difficulties of proving authentication tests, some propositions on the authentications test library were not completed. These lemmas are verified carefully in paper proofs. However, proving remaining lemmas in Coq will strengthen the correctness of authentication tests.

Bibliography

[1] The coq reference manual. https://coq.inria.fr/distrib/current/refman/, 2009.

[2] Zanella Bguelin Barthe Gilles, Grgoire Benjamin. swmath website. http://www.swmath.org/software/9443, March 2015.

[3] Yves Bertot and Pierre Castéran. Coqart. *by Springer-Verlag*, 2004.

[4] Sebastien Briais. A formalization of spi calculus in Coq. http://sbriais.free.fr/talks/talk_msr.pdf, November 2007. A talk in INRIA-Microsoft Research, Orsay, FRANCE.

[5] F Javier Thayer Fábrega, Jonathan C Herzog, and Joshua D Guttman. Strand spaces: Proving security protocols correct. *Journal of computer security*, 7(2):191–230, 1999.

[6] Joshua D Guttman and F Javier Thayer. Authentication tests and the structure of bundles. *Theoretical computer science*, 283(2):333–380, 2002.

[7] INRIA Sophia-Antipolis Mditerrane IMDEA Software Institute. Certicrypt website. http://certicrypt.gforge.inria.fr/, March 2015.

[8] Christine Paulin-Mohring. Introduction to the coq proof-assistant for practical software verification. In *Tools for Practical Software Verification*, pages 45–95. Springer, 2012.

[9] FJ Thayer Fabrega, Jonathan C Herzog, and Joshua D Guttman. Strand spaces: why is a security protocol correct? In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 160–171. IEEE, 1998.