A Sound Abstraction from the Parsing Problem in Security Protocols

Georgios Katsoris s.n. s101029

Kongens Lyngby 2013
IMM-M.Sc.-2013-112

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk IMM-M.Sc.-2013-112

Summary

This work examines the problem of how to represent the structure of messages exchanged in security protocols. Security protocols work by exchanging information using messages. Such types of messages can be XML messages, or messages with different lengths and content.

Protocol model checking abstracts from the specific details of messages, thereby excluding some potential security flaws that are based on confusing messages of similar form. Thus, a potential security flaw is left unnoticed.

To investigate this problem, a language has been defined to precisely specify the details of the examined message structure. Second, a prototype implementation was built to perform message comparisons. The input given to the implementation is the examined message set, written in our language.

The goal is to investigate the disjointness (i.e. non-confusability) of protocol messages, using our implementation's comparison results. Such a result is important when performing parallel and vertical protocol compositions. In order to claim that such compositions are secure, message and subsequently protocol disjointness is a necessary condition to be met.

We represent two widely used protocols with our language, TLS and IKE, and discuss the overall results given by our approach and implementation.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics, and was supervised by Sebastian Alexander Mödersheim.

Lyngby, 15-October-2013

Georgios Katsoris

Acknowledgements

I would like to thank my family and friends for their constant support throughout all these years.

Special thanks goes to Henning Weiss, Simon Challet and Will Courtney for their feedback and useful comments.

And last, but definitely not least, to Sebastian Alexander Mödersheim for his endless patience, dedication and constructive guidance throughout the duration of this project.

Contents

Chapter 1

Introduction to the Problem

Security protocols have a central role in information security. The importance of security in today's applications makes the deployment of security protocols an integral part of current technologies. The robustness of a protocol today is examined with static methods (like model checking), as well as with real life deployment. In other words, there exist ways to examine an individual protocol for security. Since many applications and services today are found to employ more than one protocols in order to implement their functionality, the composition of more that one protocol requires examination of the soundness of their security.

We refer to three types of possible protocol compositions. Parallel composition occurs when we have simultaneous execution of a set of isolated protocols on the same infrastructure/hardware and communication medium. Sequential protocol composition occurs when the feed of one protocol's output is used as input to another protocol. Finally, we mention vertical composition, which we define as executing a protocol on top of another protocol. An example of vertical composition would be an application protocol like mail protocol over a secure SSL channel. However, even if an individual protocol operating in isolation is secure, this does not guarantee that a composition of secure protocols will also be secure. Thus, the security of protocol compositions needs to be investigated.

Since the possible protocol compositions are numerous, we cannot study protocol composition security by examining all possible combination cases. Instead, [9] proposes four conditions that need to be satisfied by all participating protocols, in order to have a secure composition. If these conditions are met, then all possible vertical and parallel compositions of those protocols will also be secure, including their self-composition. *Message disjointness* (i.e., distinguishability) is necessary if all four conditions are to hold. By disjoint messages/message set, we mean that all messages of the examined set are distinguishable among them, and will be uniquely interpreted as they should be, avoiding any possible confusion. Equally, a disjoint protocol is a protocol with a disjoint message set. A disjoint protocol set is a set resulting from the union of disjoint protocols.

This work aims to further aid the research done on vertical protocol composition, by aiming to answer whether an examined protocol's messages are disjoint from those of other examined protocols, and from the protocol's self composition. In order to examine the disjointness of a protocol/protocols message set, we introduce a language aimed to represent protocol messages and their byte values. We also built a tool that parses messages written in our language, and compares them in order to decide whether they are disjoint. The aforementioned message comparison performed by the tool is static, and performs byte-by-byte value comparisons when possible on the examined protocol messages. The implementation goal was to create a tool that reports correctly when messages are distinguishable. We also examine the disjointness of the message sets of TLS and IKE, by expressing them with our language, and comparing them with our tool.

1.1 Preliminaries

We introduce a few definitions that will be used in this section, as well as in future sections:

a-renaming : The process of assigning values to the variable valued fields of a message. When a-renaming the message set of a protocol, we assign values to the variable fields of the messages, so that these fields have disjoint values.

Message Patterns of protocol P (MP(P)): All the sent and received messages of a protocol P. All messages of MP(P) are a-renamed.

Sub Message Patterns of protocol P (SMP(P)): We denote as ST(P) the a-renamed non-atomic fields of MP(P). We denote SMP(P) as SMP(P) : ST(P) $\bigcup$ MP(P).
To further comprehend the problem, we introduce two important notions: a

protocol disjoint from its encryptions, and pairwise disjoint protocols, as defined in [9, Def. 6]. If these two conditions are met for a set of protocols, then all protocol messages of that set are disjoint.

- **Protocol disjoint from its encryptions**
  If P is a protocol, we define its encrypted message patterns EPM(P) as: EMP(P) = a-rename(EMP_n(P)), where n $\in \mathbb{N}$, EMP_0(P) = MP(P), and EMP_{n+1}(P) = a-rename(scrypt(K_n, EMP_n(P))).

  Note that we a-rename the variables in every encryption run. A protocol P is disjoint from its encryptions if and only if there is no unifier among any of the members of EMP_i and EMP_j, meaning that the sets EMP_i and EMP_j are disjoint, for every i $\neq$ j. Practically, this means that added layers of encryption on the messages of P do not make them confusable.

  An example of a protocol disjoint from its encryptions is the following:

  Consider we have a protocol P with only one message *m1 : encr_tag*, $\{|tag, NA|\}symm$, where *encr_tag* $\neq$ *tag*. Following the notation of [9] we would write it as *m1 : encr_tag*, $scrypt(K, (tag, NA))$, but we will use below the syntax defined for our language. EMP_0 = MP(P), so the sets $EMP_{0,1}$ are:

  $EMP_0 = \{(encr\_tag, \{|tag, NA_0|\}symm_0)\}$
  $EMP_1 = \{(encr\_tag, \{|encr\_tag, \{|tag, NA_0|\}symm_0|\}symm_1)\}$

  We examine the SMP of EMP_0 and EMP_1: { (tag, NA_0) } and { (encr_tag, {| tag, NA_0 |}$symm_0$) } respectively. We see that $((tag, NA_0))\sigma$ $\neq$ $((encr\_tag, \{|tag, NA_0|\}symm_0))\sigma$ because of the constants tag $\neq$ encr_tag. So, since the encrypted contents of the elements that belong to the sets $EMP_{0,1}$ are disjoint, the sets EMP_0, EMP_1 are also disjoint.In this example, EMP_i and EMP_j are disjoint for i $\neq$ j, so the protocol P is disjoint from its encryptions.

- **Pairwise Disjoint Protocol Set**
  Let EST(P) be the non-atomic subterms of EMP(P), a-renamed, where as before: EMP(P) $\subseteq$ EST(P). The keyword subterm is equal to the notion of submessage. A set of protocols P_{set} is pairwise disjoint if every protocol P $\in$ P_{set} is disjoint from its encryptions, and for a pair P_1, P_2 $\in$ P_{set}, there exists no unifier between any element of $EST(P_1)$ and $EST(P_2)$., meaning that $EST(P_1)$ and $EST(P_2)$ are disjoint.

  A basic requirement is that all messages are non-atomic, i.e. are composed of more than one term.

We now see the direct link between message distinguishability and safe protocol compositions. In order for a protocol P to fulfill the first criterion of protocol pairwise disjointness, it must be disjoint from its own encryptions. This is achievable only if all messages of P are distinguishable among them. In order to fulfill the second criterion of protocol disjointness, $EST(P_i)$ and $EST(P_j)$ for i $\neq$ j must be disjoint. For this to hold, all messages of P_i and P_j must necessarily be distinguishable, since messages of $P_i \subseteq \mathrm{EST}(P_i)$. So we need to check that the messages of a protocol P_1 are distinguishable among them, and then examine the confusability of this protocol P_1 with the other protocols P_2, P_3 , ... that will participate in the composition.

The practical benefit of a composition of disjoint protocols is distinguishability of the transmitted messages. When using a composition of protocols from the disjoint protocol set P_{set}, for a given message M that is sent/received through the protocol stack we can infer to which protocol of the P_{set} set it belongs, and we can infer the depth of the stack of channels it went through during transmission, even if the channels are numerous and include different layers of encryption [9, p. 3].

1.2 Message Types

The notion of a field's type is to indicate the type of data this field represents. For example, a 32 bit field can be representing a 32bit long nonce, thus its type would be nonce. Formally, we would note the type of a field f as $f : \tau$. Throughout this thesis, we use the following set of basic types : *agent, nonce, key, tag, var*, which is a mix among the types used in [9], [8]. A composed type occurs from the list of basic types that it is composed of. For example, a message with a tag and a nonce would have the composed type *tag, nonce.* Functions also constitute composed types, with a formal notation of f(f_1, f_2, ... , f_n) : f(τ_1, τ_2, ... , τ_n). In this work, the functions that we use are encryption functions.

When a protocol or message is ***type-safe***, all the types present in it are interpreted correctly. Equally, this means that type-interpretation cannot be manipulated, and type-confusion doesn't exist. Thus, we see that by having message disjointness in a protocol, we also achieve type-safety for the messages of that protocol, since every message is interpreted correctly. The notion of ***type-safe*** protocol/message is equal to the ***well-typed*** protocol/message, as used in [9, p. 18], [8, sec. 3.1].

1.3 Protocol Model Checking

When model checking a protocol for security, we abstract from the very specific details involved in each message such as field sizes, and use a more abstract notion. For example, a message like *f_1 : tag, agent_ID[64], nonce_A[32]* could be written into something more abstract like *f_1(A,NA)* , if fed to a model checking tool. However, this abstraction should happen only if we know that *f_1* is distinguishable from all other messages. This would also mean that *f_1* is type-safe because of its distinguishability, thus its field types cannot be misinterpreted as of another type.

Type-safe protocols have the advantage of reducing the resulting state-space when model checking them for their security, because we examine only well typed runs of the protocol [8, p. 1]. Moreover, an attack can happen on a type-safe protocol only if it is a well typed attack [8, p. 1],[9, p. 14].

1.4 Achieving Message Disjointness

In order to achieve message disjointness, one approach would be to implement a tagging scheme, so that each message has embedded the information (in the form of a tag) that will distinguish it from other messages. For existing protocols that do not incorporate this approach, it is important to perform a message disjointness check, to see whether a protocol's messages are already disjoint, or not.

1.5 Where This Thesis Contributes

The goal of this thesis is to create a language and a tool for examining protocol disjointness. This check includes the messages of an individual protocol, as well as all the messages of a set of protocols. A key element for achieving this is the introduction of the language that we utilize, in order to represent a protocol and process it further. The aim was double: to create a language that would represent messages found in traditional message passing, XML and padding protocols, and to create a tool that would be able to compare all these different types of messages among them. The focus was on describing and examining protocols of current and widespread use.

The goal is to statically describe messages and their fields, and perform the message comparisons in a static way as well, without thinking in terms of protocol execution. Although the static approach can be limiting in some aspects of protocol examination, it relieves us of the burden of actually running the protocol. Moreover, different implementations of the same protocol can differ in some aspects of a protocol's behavior as in [4, p. 59], while the static approach is universal. The drawback of static analysis is that it can be limiting in some aspects, where dynamic analysis could be more beneficial.

The language we define takes into consideration the ideas and data structures found in [4],[6], which were the two protocols we examined thoroughly. In has to be noted that some guidelines that these protocols follow might not fit other protocols. For example, both TLS and IKE always have an encrypted field as the last field of a message, and no message can have more than one encrypted fields. This might not suite other protocols though. Because we could not look at all possible cases, we believe there is still elements to add, in order to have a truly universal protocol message language.

Our aim is to have an implementation that accepts as input the set of messages we defined using our language, and compares them in order to conclude if they are disjoint or not. Although there is always room for enriching an implementation with more capabilities and features, our implementation manages the aforementioned goal for the message cases we examined.

1.6 Development Tools

Because this work requires processing of text, we had to use a lexical analyzer and a parser generator in combination. A lexical analyzer converts the sequence of input characters/strings it is fed into a sequence of tokens. This token sequence is then fed to a parser, which applies the corresponding semantic action depending on the input. In order to obtain a parser we have to generate one, by feeding a grammar specification (in the form of a text file) into a tool called parser generator.

For the checker's implementation, we used the Haskell programming language [11]. For lexical analysis, we used the Alex lexical analyzer generator [12], and the Happy parser generator [13] was used for generating a parser for our language. Alex and Happy can work in combination. The compiler used was GHC, and the used OS was a 64 bit Ubuntu Linux version 13.04.

Chapter 2

Context Free Languages and Grammar

We discuss briefly here the concept of a language and grammar. In this report, we will only discuss about context free grammars. A grammar defines a language [1, p169], and if a language is derived from a CFG, then it is a context free language (CFL) [1, p117].

The concrete syntax of a language describes the proper form that strings belonging to the language should have, while the semantics of the language defines what these strings mean, i.e. what they represent.

We distinguish among concrete and abstract syntax of a language. Concrete syntax specifies how the language expressions must look like. Abstract syntax is represented by abstract syntax trees or simply syntax trees, which represent the hierarchical syntactic structure of the input strings [10, ch 2.1]. An abstract syntax tree is a data structure that is used to hold the significant parts of the parsed expressions. A significant token could for example be placed as the root element of the syntax tree, less significant tokens as leaves e.c.t.

A context-free grammar is defined by four components:

- Terminal symbols or terminals, that form the strings of the language, noted as the set T

- A finite set of variables, also called non-terminals, noted V. Each variable represents a language, i.e. a set of strings.
- A starting symbol, called S, where S is part of the variable set V. This variable represents the language being defined.
- A finite set of productions (or rules) P that represent the recursive definition of the language. Each production consists of a variable, the production symbol $\rightarrow$, and a string of zero or more terminals and variables (also called the body of the production).

So, a context-free grammar (CFG) is defined as G(V,T,P,S). The language that derives from G is noted as L(G).

In order to derive the language of a context-free grammar, we initially expand all the productions whose head is the start symbol S. We further expand the resulting string by further using all available productions, until we reach a string consisting of terminal characters only. This process is called *derivation*, and symbolized with =>. The set of all strings consisting of terminal characters, obtained in the above way, is called the language of the grammar. Formally, the above process is noted as L(G) = { w $\in$ T | S => G w }.

A formal grammar is considered "context free" when its production rules can be applied regardless of the context of a nonterminal. It does not matter which symbols the nonterminal is surrounded by, the single nonterminal on the left hand side can always be replaced by the right hand side. The grammar that we use for our defined language has exactly this functionality, thus we use a context free grammar.

Next to a grammar production we note a semantic action, which is executed when the production is executed. Program fragments embedded within production bodies are called semantic actions [10, ch.2]. The executed code of the semantic action must belong to our used abstract syntax, and the execution of semantic actions constructs the abstract syntax tree.

We now show a short part of our grammar in a simplified form in Figure 2.1. For a complete reference, we direct the reader to the grammar file, found in the file my_parser.y, since some appearing tokens are not mentioned further here.

A code snippet from our Happy grammar file is seen in Figure 2.2. The left part is the grammar production, and the right part is the semantic action, which contains the corresponding abstract syntax. On the left part of the expression, the tokens contained between the " symbols are terminal symbols, the *cap_ident,*

```
MsgBody ::= empty
        | MsgField
        | MsgField , MsgBody

MsgField ::= identifier( MsgBody )
          | identifier
          | cap_ident
          | [ MsgField ]
          | [ MsgBody ]
          | MsgField*
          | (MsgBody)*
          | MsgField+
          | (MsgBody)+
          | {| MsgBody |}symm(cap_ident,cap_ident)
          | { MsgBody }public(cap_ident)
          | { MsgBody }priv(cap_ident)
```

Figure 2.1: Part of our used grammar, expressed in simplified form.

```
MsgBody :: { [Field] }
--Same as MsgField+ : Left recursion (p7,Happy manual)
MsgBody : 'empty'              { [] }
        | MsgField             { [$1] }
        | MsgField ',' MsgBody   { $1 : $3 }

MsgField : identifier '(' MsgBody ')'          { ParamField $1 $3 }
         | identifier                          { PlainField $1 }
         | cap_ident                           { Param $1 "m_field" }
         | '[' MsgField ']'                    { RepeatedField [$2] R_0_1 }
         | '[' MsgBody ']'                     { RepeatedField $2 R_0_1 }
         | MsgField '*'                        { RepeatedField [$1] R_0_Inf }
         | '(' MsgBody ')' '*'                 { RepeatedField $2 R_0_Inf }
         | MsgField '+'                        { RepeatedField [$1] R_1_Inf }
         | '(' MsgBody ')' '+'                 { RepeatedField $2 R_1_Inf }
         | '{''|' MsgBody '|''}''symm''('cap_ident','cap_ident')'
             { EncryptedField $3 (Symm (Param $8 "agent") (Param $10 "agent")) }
         | '{' MsgBody '}' 'public' '(' cap_ident ')'
             { EncryptedField $2 (Public (Param $6 "agent")) }
         | '{' MsgBody '}' 'priv' '(' cap_ident ')'
             { EncryptedField $2 (Priv (Param $6 "agent")) }
```

Figure 2.2: Part of our used grammar file. The corresponding semantic action is noted on the right of each production.

identifier symbols are tokens carrying a string of capital and lowercase characters respectively. The rest of the appearing symbols are non-terminals.

Chapter 3

Algorithm Description and Proof

We discuss the functionality of our implementation by presenting the algorithm it implements. The examined field cases are some (but not all) of the cases included in our syntax & implementation. But more field cases are easy to add in the presented scheme below. The presented algorithm does not follow the implementation's behavior exactly. However, their overall functionality is the same.

We present the possible field comparison cases, and include a proof for each case. The proof is by structural induction over the recursive structure of the procedure. Mathematical induction works by using a statement for the smaller problem, and implying that statement when examining the bigger problem. We prove each case under the inductive assumption that in all recursive algorithm calls, the procedure already works correctly. Examining the algorithm's steps below, we can see that the recursive calls work on smaller inputs each time. This guarantees that the algorithm terminates, and the problem's size decreases on each recursive call. Structural induction is an induction process over recursive structures, and because our algorithm is recursive, it is a proper proof strategy.

3.1 Checker Algorithm and Proof of Each Case

Consider we are to compare two formats $F1, F2$, each of them being composed of fields, where $F_1 = f_1,\ f_2,\ \ldots\ ,\ f_n$ and $F_2 = f'_1,\ f'_2,\ \ldots\ ,\ f'_m$. A format F can also be the empty word ϵ only, in which case $[\![\ \mathrm{F}\ ]\!] = [\![\ \epsilon\ ]\!]$. In our case, the empty word is a bytestring of zero length, and $[\![\ \epsilon\ ]\!] = \{\ \epsilon\ \}$, which is not equal to the empty set. For ϵ applies the $\epsilon \cdot f = f$ property, where $\cdot$ here is used as a bytestring concatenation operator.We consider the fields below, where c is a constant of one byte, $x[l]$ a variable of length l bytes, $x[[l]]$ a vector with a length field l bytes long, and a payload of 0 to 256^l bytes. Note that in the case of vectors here, we use a more generic vector syntax which covers the largest possible payload size that length l can have, while we also simplify the vector notation from the usual $x[[l]] < a..b >$ syntax. But since $[\![x[[l]] < a..b >]\!] \subseteq [\![x[[l]]]\!]$, the algorithm would work with a smaller vector as well. We also use the $x(Rest)$ notation to represent an arbitrary length field, which can also be the empty bytestring.

Formally we specify the above as c',c $\in \mathbb{B}$, where $\mathbb{B} = \{\ 0..255\ \}$, denoting the values of a 1 byte long string. We define as $\mathbb{B}^*$ as the set of all byte strings, including the empty word. $l \in \mathbb{N}$, where $0 \in \mathbb{N}$. We use the $\times$ operator as the string concatenation operator in sets of strings. Formally, we would write is as: $A \times B = \{a \cdot b | a \in A, b \in B\}$, where $\cdot$ is the string concatenation operator.

The used grammar is seen below:
f ::= c | x[l] | x[[l]] | x(Rest)

$F ::= f_1, f_2, ..., f_n$, where $n \geq 0$. If $n = 0$, $F = \epsilon$.

The semantics function for the above syntax follows:
$[\![\epsilon]\!] = \{\ \epsilon\ \}$
$[\![c]\!] = \{\ \mathrm{c}\ \}$
$[\![x[l]]\!] = \mathbb{B}^l$, where $l \geq 0$
$[\![x[[l]]]\!] = \{\ \mathrm{uv}\ |\ \mathrm{u} \in \mathbb{B}^l,\ \mathrm{v} \in \mathbb{B}^u\ \}$, where the value of u is used as payload length indicator. Note that $l \geq 1$ here.
$[\![x(Rest)]\!] = [\![x'[1]]\!] \times [\![x'(Rest)]\!] \cup \{\epsilon\}$
$[\![f_1, f_2, ..., f_n]\!] = [\![f_1]\!] \times \ldots \times [\![f_n]\!]$

Our concrete syntax allows the declaration of arrays with zero index, and vectors with no payload data. Thus, we have: $x[0] == \epsilon$ because $x[0]$ is an empty bytestring. Note that "0" $\cdot\, \epsilon \in [\![x[[l]]]\!]$, which is the empty payload case, and the length indicator's value is zero.

We use the word disj to state that two formats are disjoint, and non-disj to state that they are not disjoint. We use the operators $\vee, \wedge$ with the disj,non-disj values, as defined in the array below:

A	B	A $\wedge$ B	A $\vee$ B
disj	disj	disj	disj
disj	nondisj	nondisj	disj
nondisj	disj	nondisj	disj
nondisj	nondisj	nondisj	nondisj

We examine the cases of comparing two empty formats, an empty and a non-empty format, and finally two non-empty formats. If two formats are confusable, we return the string disjoint (**disj**), otherwise return **nondisj**. The checker has the symmetric property, meaning:
$check(f_1, f_2, ..., f_n; f'_1, f'_2, ..., f'_n) = check(f'_1, f'_2, ..., f'_n; f_1, f_2, ..., f_n)$. Note that if the **nondisj** result appears once in one of the *check* instances of the $\bigwedge$ *check* below, the entire statement is equal to **nondisj**. In the algorithm below, we make comments with the // symbol, followed by text. After each algorithm case, noted with the $\triangleright, \bullet$, follows the case proof.

Checker Algorithm

- $check(\epsilon; \epsilon) =$ **nondisj**

- $check(\epsilon; f_1, f_2, ..., f_n) = \bigvee_{i=1}^{n} check(\epsilon, f_i)$

  Given that $check(\epsilon, f_1, f_2, ..., f_n) = disj$, we want to show that
  $[\![\epsilon]\!] \bigcap [\![f_1, f_2, ..., f_n]\!] = \varnothing$.
  By the algorithm we thus have
  $\Rightarrow \bigvee_{i=1}^{n} check(\epsilon, f_i) = disj$
  $\Rightarrow$ for at least one i $\in$ { 1..n }, $check(\epsilon, f_i) = disj$ holds.
  $\Rightarrow [\![\epsilon]\!] \bigcap [\![f_i]\!] = \varnothing$ for at least one i $\in$ { 1..n }.
  $\Rightarrow [\![\epsilon]\!] \bigcap ([\![f_1]\!] \times [\![f_2]\!] \times ... \times [\![f_n]\!]) = \varnothing$
  $\Rightarrow [\![\epsilon]\!] \bigcap [\![f_1, f_2, ..., f_n]\!] = \varnothing$

- $check(f_1, f_2, ..., f_n; f'_1, f'_2, ..., f'_n)$
  case(f_1, f'_1) of : // Algorithm is symmetric

  $\triangleright$ (c,c') = **if**(c/=c') **then** disj
  else $check(f_2, ..., f_n; f'_2, ..., f'_n)$
  Given that $check(c, f_2, ..., f_n; c', f'_2, ..., f'_m) = disj$, we want to show that

$[\![c, f_2, ..., f_n]\!] \bigcap [\![c', f'_2, ..., f'_m]\!] = \varnothing$. We examine both algorithm cases:
1st case: c /= c'
Thus, $[\![c]\!] \bigcap [\![c']\!] = \varnothing$
$\Rightarrow [\![c]\!] \times ([\![f_2]\!] \times ... \times [\![f_n]\!]) \bigcap [\![c']\!] \times ([\![f'_2]\!] \times ...[\![f'_m]\!]) = \varnothing$, this holds regardless of applying at the same time both $[\![f_2]\!] \times ... \times [\![f_n]\!]$, $[\![f'_2]\!] \times ...[\![f'_m]\!]$ as prefixes or suffixes.
$\Rightarrow [\![c, f_2, ..., f_n]\!] \bigcap [\![c', f'_2, ..., f'_m]\!] = \varnothing$
2nd case: c == c'
Using the initial assumption and by the algorithm, we have that the statement $check(f_2, ..., f_n; f'_2, ..., f'_m) = disj$ holds.
$\Rightarrow [\![f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing$
$\Rightarrow [\![c]\!] \times [\![f_2, ..., f_n]\!] \bigcap [\![c']\!] \times [\![f'_2, ..., f'_m]\!] = \varnothing$, because $[\![c]\!] \bigcap [\![c']\!] \neq \varnothing$, we do not tamper the value of the statement by applying $[\![c]\!], [\![c']\!]$ as prefix.

▷ (c,x[l]) = **if**(l==0) **then** $check(c, f_2, ..., f_n; f'_2, ..., f'_m)$
else $check(f_2, ..., f_n; x[l-1], f'_2, ..., f'_m)$
Given that $check(c, f_2, ..., f_n; x[l], f'_2, ..., f'_m) = disj$, we want to show that
$[\![c, f_2, ..., f_n]\!] \bigcap [\![x[l], f'_2, ..., f'_m]\!] = \varnothing$.
We examine both algorithm cases:
1st case: l == 0
Using the initial assumption and by the algorithm, we have that $check(c, f_2, ..., f_n; f'_2, ..., f'_m) = disj$ holds.
$\Rightarrow [\![c, f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing$
$\Rightarrow [\![c, f_2, ..., f_n]\!] \bigcap ([\![x[l]]\!] \times [\![f'_2, ..., f'_m]\!]) = \varnothing$, since $[\![x[l]]\!] = \{\epsilon\}$.
$\Rightarrow [\![c, f_2, ..., f_n]\!] \bigcap [\![x[l], f'_2, ..., f'_m]\!] = \varnothing$
2nd case: l > 0
Using the initial assumption and by algorithm, we have that $check(f_2, ..., f_n; x[l-1], f'_2, ..., f'_m) = disj$ holds.
$\Rightarrow [\![f_2, ..., f_n]\!] \bigcap [\![x[l-1], f'_2, ..., f'_m]\!] = \varnothing$
For a new variable $x'[1]$ we have that $[\![x[l]]\!] = [\![x'[1]]\!] \times [\![x[l-1]]\!]$, and the disjointness remains if we apply $[\![x'[1]]\!], [\![c]\!]$ as prefixes in the examined statement.
$\Rightarrow [\![c]\!] \times [\![f_2, ..., f_n]\!] \bigcap [\![x'[1]]\!] \times [\![x[l-1], f'_2, ..., f'_m]\!] = \varnothing$
$\Rightarrow [\![c, f_2, ..., f_n]\!] \bigcap [\![x[l], f'_2, ..., f'_m]\!] = \varnothing$

▷ (c,x[[l]]) = $\bigwedge_{i=(i_1,...,i_l)\in\mathbb{B}^l} check(c, f_2, ..., f_n; i, x'[i], f'_2, ..., f'_m)$
Given that $check(c, f_2, ..., f_n; x[[l]], f'_2, ..., f'_m) = disj$, we want to show
$[\![c, f_2, ..., f_n]\!] \bigcap [\![x[[l]], f'_2, ..., f'_m]\!] = \varnothing$.
Note that in this and subsequent appearances of $x[[l]]$, we have that $[\![x[[l]]]\!] = \bigcup_{i=(i_1,...,i_l)\in\mathbb{B}^l} \{i\} \times [\![x'[i]]\!]$.
By the algorithm, we have that

$\Rightarrow \bigwedge_{i=(i_1,...,i_l)\in\mathbb{B}^l} check(c, f_2, ..., f_n; i, x'[i], f'_2, ..., f'_m) = disj$ holds.
$\Rightarrow check(c, f_2, ..., f_n; i, x'[i], f'_2, ..., f'_m) = disj,\ \forall i \in \mathbb{B}^l$
$\Rightarrow [\![c, f_2, ..., f_n]\!] \bigcap [\![i, x'[i], f'_2, ..., f'_m]\!] = \varnothing,\ \forall i \in \mathbb{B}^l$
$\Rightarrow [\![c, f_2, ..., f_n]\!] \bigcap (\bigcup_{i=(i_1,...,i_l)\in\mathbb{B}^l} [\![i, x'[i], f'_2, ..., f'_m]\!]) = \varnothing$
The disjunction remains empty, because we are adding sets that have no intersection with $[\![c, f_2, ..., f_n]\!]$.
$\Rightarrow [\![c, f_2, ..., f_n]\!] \bigcap (\bigcup_{i=(i_1,...,i_l)\in\mathbb{B}^l} [\![i, x'[i]]\!]) \times [\![f'_2, ..., f'_m]\!] = \varnothing.$
$\Rightarrow [\![c, f_2, ..., f_n]\!] \bigcap [\![x[[l]], f'_2, ..., f'_m]\!] = \varnothing$

```
▷ (x[l],x'[l']) = if( l==l' ) then $check(f_2, ..., f_n; f'_2, ..., f'_m)$
                  else if(l>l') then
                    if(l'==0) then $check(x[l], f_2, ..., f_n; f'_2, ..., f'_m)$
                    else $check(x[l-1], f_2, ..., f_n; x'[l'-1], f'_2, ..., f'_m)$
                  else if(l<l') // Symmetry in algorithm
```

Given that $check(x[l], f_2, ..., f_n; x'[l'], f'_2, ..., f'_m) = disj$, we want to show that
$[\![x[l], f_2, ..., f_n]\!] \bigcap [\![x'[l'], f'_2, ..., f'_m]\!] = \varnothing.$
By the algorithm we have three cases, and a fourth case which applies to the symmetry of the algorithm, thus we do not have to examine it further.
1st case: l==l'
Using the initial assumption and by the algorithm, we have that $check(f_2, ..., f_n; f'_2, ..., f'_m) = disj$ holds.
$\Rightarrow [\![f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing$
Since l==l', $[\![x[l]]\!] \bigcap [\![x'[l']]\!] \neq \varnothing$, for all possible values of l.
$\Rightarrow ([\![x[l]]\!] \times [\![f_2, ..., f_n]\!]) \bigcap ([\![x'[l']]\!] \times [\![f'_2, ..., f'_m]\!]) = \varnothing$, the statement holds regardless of applying both $[\![x[l]]\!], [\![x'[l']]\!]$ as prefix or postfix.
$\Rightarrow ([\![x[l], f_2, ..., f_n]\!]) \bigcap ([\![x'[l'], f'_2, ..., f'_m]\!]) = \varnothing$
2nd case: l'==0
Using the initial assumption and by the algorithm, we have that $check(x[l], f_2, ..., f_n; f'_2, ..., f'_m) = disj$ holds.
$\Rightarrow [\![x[l], f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing$
$\Rightarrow [\![x[l], f_2, ..., f_n]\!] \bigcap ([\![x'[l']]\!] \times [\![f'_2, ..., f'_m]\!]) = \varnothing$, since $[\![x'[l']]\!] = \{\epsilon\}$.
$\Rightarrow [\![x[l], f_2, ..., f_n]\!] \bigcap [\![x'[l'], f'_2, ..., f'_m]\!] = \varnothing$
3rd case: l>l>0
Using the initial assumption and by the algorithm, we have that $check(x[l-1], f_2, ..., f_n; x'[l'-1], f'_2, ..., f'_m) = disj$ holds.
$\Rightarrow [\![x[l-1], f_2, ..., f_n]\!] \bigcap [\![x'[l'-1], f'_2, ..., f'_m]\!] = \varnothing$
We define two new variables $x''[1], x'''[1]$, thus $[\![x''[1]]\!] \bigcap [\![x'''[1]]\!] \neq \varnothing$. Applying them as prefixes to the above statement does not alter the disjointness result.

$\Rightarrow ([\![x''[1]]\!] \times [\![x[l-1], f_2, ..., f_n]\!]) \bigcap ([\![x'''[1]]\!] \times [\![x'[l'-1], f'_2, ..., f'_m]\!]) = \varnothing$
Since $[\![x''[1]]\!] \times [\![x[l-1]]\!] = [\![x[l]]\!]$ and equally for $x'[l']$, we have that:
$\Rightarrow ([\![x''[1]]\!] \times [\![x[l-1], f_2, ..., f_n]\!]) \bigcap ([\![x'''[1]]\!] \times [\![x'[l'-1], f'_2, ..., f'_m]\!]) = \varnothing$
$\Rightarrow [\![x[l], f_2, ..., f_n]\!] \bigcap [\![x'[l'], f'_2, ..., f'_m]\!] = \varnothing$
4th case: The fourth case is already solved, because of the symmetry of the algorithm.

▷ (x[l],x'[[l']]) = **if** (l==0) **then** $\mathit{check}(f_2, ..., f_n; x'[[l']], f'_2, ..., f'_m)$
else $\bigwedge_{i=(i_1..i_l)\in\mathbb{B}^l} \mathit{check}(x[l], f_2, ..., f_n; i, x'[i], f'_2, ..., f'_m)$
Given that $\mathit{check}(x[l], f_2, ..., f_n; x'[[l']], f'_2, ..., f'_m) = disj$, we want to show that
$[\![x[l], f_2, ..., f_n]\!] \bigcap [\![x'[[l']], f'_2, ..., f'_m]\!] = \varnothing$.
Following the algorithm steps, we have two cases:
1st case: l==0
Using the initial assumption and by the algorithm, we have that $\mathit{check}(f_2, ..., f_n; x'[[l']], f'_2, ..., f'_m) = disj$ holds.
$\Rightarrow [\![f_2, ..., f_n]\!] \bigcap [\![x'[[l']], f'_2, ..., f'_m]\!] = \varnothing$
$\Rightarrow ([\![x[0]]\!] \times [\![f_2, ..., f_n]\!]) \bigcap [\![x'[[l']], f'_2, ..., f'_m]\!] = \varnothing$, since $[\![x[0]]\!] = \{\epsilon\}$
$\Rightarrow ([\![x[l]]\!] \times [\![f_2, ..., f_n]\!]) \bigcap [\![x'[[l']], f'_2, ..., f'_m]\!] = \varnothing$
$\Rightarrow [\![x[l], f_2, ..., f_n]\!] \bigcap [\![x'[[l']], f'_2, ..., f'_m]\!] = \varnothing$
2nd case:
Using the initial assumption and by the algorithm, we have that
$\bigwedge_{i=(i_1..i_{l'})\in\mathbb{B}^{l'}} \mathit{check}(x[l], f_2, ..., f_n; i, x'[i], f'_2, ..., f'_m) = disj$ holds.
This means that $\mathit{check}(x[l], f_2, ..., f_n; i, x'[i], f'_2, ..., f'_m) = disj$, $\forall i \in \mathbb{B}^{l'}$ holds
$\Rightarrow [\![x[l], f_2, ..., f_n]\!] \bigcap ([\![i, x'[i]]\!] \times [\![f'_2, ..., f'_m]\!]) = \varnothing$, $\forall i \in \mathbb{B}^{l'}$
By merging all the i statements, we have:
$\Rightarrow [\![x[l], f_2, ..., f_n]\!] \bigcap ((\bigcup_{i=(i_1..i_{l'})\in\mathbb{B}^{l'}} [\![i, x'[i]]\!]) \times [\![f'_2, ..., f'_m]\!]) = \varnothing$
$\Rightarrow [\![x[l], f_2, ..., f_n]\!] \bigcap [\![x'[[l']], f'_2, ..., f'_m]\!] = \varnothing$

▷ (x[[l]],x'[[l']]) = $\bigwedge_{\substack{i=(i_1,...,i_l)\in\mathbb{B}^l \\ i'=(i'_1,...,i'_{l'})\in\mathbb{B}^{l'}}} \mathit{check}(i, x[i], f_2, ..., f_n; i', x'[i'], f'_2, ..., f'_m)$
Given that:
$\mathit{check}(x[[l]], f_2, ..., f_n; x'[[l']], f'_2, ..., f'_m) = disj$, we want to show
$[\![x[[l]], f_2, ..., f_n]\!] \bigcap [\![x'[[l']], f'_2, ..., f'_m]\!] = \varnothing$
By the algorithm, we have that
$\bigwedge_{\substack{i=(i_1,...,i_l)\in\mathbb{B}^l \\ i'=(i'_1,...,i'_{l'})\in\mathbb{B}^{l'}}} \mathit{check}(i, x[i], f_2, ..., f_n; i', x'[i'], f'_2, ..., f'_m) = disj$ also holds.
$\Rightarrow \mathit{check}(i, x[i], f_2, ..., f_n; i', x'[i'], f'_2, ..., f'_m) = disj$, $\forall i \in \mathbb{B}^l, \forall i' \in \mathbb{B}^{l'}$.
$\Rightarrow [\![i, x[i], f_2, ..., f_n]\!] \bigcap [\![i', x'[i'], f'_2, ..., f'_m]\!] = \varnothing$, $\forall i \in \mathbb{B}^l, \forall i' \in \mathbb{B}^{l'}$. By merging all statements, we have that:

$\Rightarrow (\bigcup_{i=(i_1,...,i_l)\in\mathbb{B}^l} [\![i, x[i], f_2, ..., f_n]\!]) \bigcap (\bigcup_{i'=(i'_1,...,i'_{l'})\in\mathbb{B}^{l'}} [\![i', x'[i'], f'_2, ..., f'_m]\!]) =$
$\varnothing$
$\Rightarrow (\bigcup_{i=(i_1,...,i_l)\in\mathbb{B}^l} [\![i, x[i]]\!]) \times [\![f_2, ..., f_n]\!] \bigcap (\bigcup_{i'=(i'_1,...,i'_{l'})\in\mathbb{B}^{l'}} [\![i', x'[i']]\!]) \times [\![f'_2, ..., f'_m]\!]) =$
$\varnothing$
$\Rightarrow [\![x[[l]], f_2, ..., f_n]\!] \bigcap [\![x'[[l']], f'_2, ..., f'_m]\!] = \varnothing$

▷ (c,x(Rest)) = $check(f_2, ..., f_n; f'_2, ..., f'_m)$ // len(x(Rest))==1
$\bigwedge check(f_2, ..., f_n; x'(Rest), f'_2, ..., f'_m)$ //len(x(Rest))>1
$\bigwedge check(c, f_2, ..., f_n; f'_2, ..., f'_m)$ //x(Rest) is empty

Given that $check(c, f_2, ..., f_n; x(Rest), f'_2, ..., f'_m) = disj$, we want to show
$[\![c, f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!] = \varnothing$
By the algorithm, we have that:
$check(f_2, ..., f_n; f'_2, ..., f'_m) = disj$,
$check(f_2, ..., f_n; x'(Rest), f'_2, ..., f'_m) = disj$ and
$check(c, f_2, ..., f_n; f'_2, ..., f'_m) = disj$ all hold
$\Rightarrow [\![f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing$ **A**,
$[\![f_2, ..., f_n]\!] \bigcap [\![x'(Rest), f'_2, ..., f'_m]\!] = \varnothing$ **B**,
$[\![c, f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing$ **C**
where all **A,B,C** statements hold. Note that we express $[\![x(Rest)]\!]$ as $[\![x(Rest)]\!] = [\![x'[1]]\!] \times [\![x'(Rest)]\!] \cup \{\epsilon\}$, as by the definition.
We prove by contradiction that $[\![c, f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!] = \varnothing$. Suppose there is some string
s $\in \mathbb{B}^*$: s $\in ([\![c, f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!])$, meaning that the intersection is not empty. Then, at least one of the following statement sets **1,2,3** should hold (and all included statements in that statement set must all hold), covering all length cases of x(Rest):
$(s = cs', s' \in [\![f_2, ..., f_n]\!], s' \in [\![f'_2, ..., f'_m]\!])$ **1**, when length of x(Rest) is equal to length of c.
$(s = cs', s' \in [\![f_2, ..., f_n]\!], s' \in [\![x'(Rest), f'_2, ..., f'_m]\!])$ **2**, when length of x(Rest) > length of c
$(s \in [\![c, f_2, ..., f_n]\!], s \in [\![f'_2, ..., f'_m]\!])$ **3**, when x(Rest) is the empty bytestring
From statement **1**, $[\![f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] \neq \varnothing$, which contradicts statement **A** which holds, thus **1** does not hold.
From statement **2**, $[\![f_2, ..., f_n]\!] \bigcap [\![x'(Rest), f'_2, ..., f'_m]\!] \neq \varnothing$, which contradicts statement **B** which holds, thus **2** does not hold.
From statement **3**, $[\![c, f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] \neq \varnothing$, which contradicts statement **C** which holds, thus **3** does not hold.
Since none of **1,2,3** hold, $[\![c, f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!] = \varnothing$.

▷ (x[l],x(Rest)) = **if**(l==0) **then** $check(f_2, ..., f_n; x(Rest), f'_2, ..., f'_m)$

```
            else $check(f_2,...,f_n;f'_2,...,f'_m)$//len(x(Rest))==l
              $\bigwedge check(f_2,...,f_n;x'(Rest),f'_2,...,f'_m)$ //len(x(Rest))>l
              $\bigwedge_{i=1}^{l-1} check(x[i],f_2,...,f_n;f'_2,...,f'_m)$ //len(x(Rest))<l
              $\bigwedge check(x[l],f_2,...,f_n;f'_2,...,f'_m$ //len(x(Rest))==0
```

Given that $check(c, f_2, ..., f_n; x(Rest), f'_2, ..., f'_m) = disj$, we want to show
$[\![x[l], f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!] = \varnothing$
By the algorithm, we have two cases :
1st case: l == 0
Following the algorithm and using the initial given statement, we have that
$check(f_2, ..., f_n; x(Rest), f'_2, ..., f'_m) = disj$
$\Rightarrow [\![f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!] = \varnothing$
$\Rightarrow [\![x[l]]\!] \times [\![f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!] = \varnothing$, since $[\![x[l]]\!] = \{\epsilon\}$
$\Rightarrow [\![x[l], f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!] = \varnothing$
2nd case: l > 0
Following the algorithm and using the initial given statement, we have that
$check(f_2, ..., f_n; f'_2, ..., f'_m) = disj$,
$check(f_2, ..., f_n; x'(Rest), f'_2, ..., f'_m) = disj$,
$\bigwedge_{i=1}^{l-1} check(x[i], f_2, ..., f_n; f'_2, ..., f'_m) = disj$,
$check(x[l], f_2, ..., f_n; f'_2, ..., f'_m = disj$ all hold
$\Rightarrow [\![f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing$ **A**
$[\![f_2, ..., f_n]\!] \bigcap [\![x'(Rest), f'_2, ..., f'_m]\!] = \varnothing$ **B**
$[\![x[i], f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing, \forall i\{1..(l-1)\}$ **C**
$[\![x[l], f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing$ **D**
Note that we express $[\![x(Rest)]\!]$ as $[\![x(Rest)]\!] = [\![x'[l]]\!] \times [\![x'(Rest)]\!] \cup \{\epsilon\}$, as by the definition.
For the second case, we prove by contradiction that $[\![x[l], f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!] = \varnothing$. Suppose there is some string
$s \in \mathbb{B}^* : s \in ([\![x[l], f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!])$, meaning that the intersection is not empty. Then, at least one of the following **1,2,3,4** statement sets must hold (and all included statements in that statement set must all hold), which cover all length cases of x(Rest):
$(s = x[l] \cdot s', s' \in [\![f_2, ..., f_n]\!], s' \in [\![f'_2, ..., f'_m]\!])$ **1**, when x(Rest) has length l
$(s = x[l] \cdot s', s' \in [\![f_2, ..., f_n]\!], s' \in [\![x'(Rest), f'_2, ..., f'_m]\!])$ **2**, when x(Rest) has length > l
$(s = x[j] \cdot x[i] \cdot s', i + j = l, s_i = x[i] \cdot s', \exists i \in \{1..l-1\} : s_i \in [\![x[i], f_2, ..., f_n]\!], s_i \in [\![f'_2, ..., f'_m]\!])$ **3**, when x(Rest) has length < l

$(s = x[l] \cdot s', s \in [\![x[l], f_2, ..., f_n]\!], s \in [\![f'_2, ..., f'_m]\!])$ **4**, when x(Rest) has length = 0
From statement **1**, $[\![f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] \neq \varnothing$, which contradicts statement **A** which holds. Thus, **1** does not hold.
From statement **2**, $[\![f_2, ..., f_n]\!] \bigcap [\![x'(Rest), f'_2, ..., f'_m]\!] \neq \varnothing$, which contradicts statement **B** which holds. Thus, **2** does not hold.
From statement **3**, $[\![x[i], f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] \neq \varnothing$ for at least one i, which contradicts statement **C** which holds. Thus, **3** does not hold.
From statement **4**, $[\![x[l], f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] \neq \varnothing$, which contradicts statement **D** which holds. Thus, **4** does not hold.
Since none of **1,2,3,4** hold, $[\![x[l], f_2, ..., f_n]\!] \bigcap [\![x(Rest), f'_2, ..., f'_m]\!] = \varnothing$.

▷ (x[[l]],x'(Rest)) = $\bigwedge_{i=(i_1..i_l)\in\mathbb{B}^l} check(i, x[i], f_2, ..., f_n; x'(Rest), f'_2, ..., f'_m)$

Given that $check(x[[l]], f_2, ..., f_n; x'(Rest), f'_2, ..., f'_m) = disj$, we want to show
$[\![x[[l]], f_2, ..., f_n]\!] \bigcap [\![x'(Rest), f'_2, ..., f'_m]\!] = \varnothing$
By the algorithm, we have that
$\bigwedge_{i=(i_1..i_l)\in\mathbb{B}^l} check(i, x[i], f_2, ..., f_n; x'(Rest), f'_2, ..., f'_m) = disj$ holds
$\Rightarrow [\![i, x[i], f_2, ..., f_n]\!] \bigcap [\![x'(Rest), f'_2, ..., f'_m]\!], \forall i \in \mathbb{B}^l$
$\Rightarrow (\bigcup_{i=(i_1..i_l)\in\mathbb{B}^l} [\![i, x[i], f_2, ..., f_n]\!]) \bigcap [\![x'(Rest), f'_2, ..., f'_m]\!]$
$\Rightarrow (\bigcup_{i=(i_1..i_l)\in\mathbb{B}^l} [\![i, x[i]]\!]) \times [\![f_2, ..., f_n]\!] \bigcap [\![x'(Rest), f'_2, ..., f'_m]\!]$
$\Rightarrow [\![x[[l]], f_2, ..., f_n]\!] \bigcap [\![x'(Rest), f'_2, ..., f'_m]\!] = \varnothing$

▷ (x_1(Rest),x_2(Rest)) = $check(f_2, ..., f_n; f'_2, ..., f'_m)$
//len. of x_1(Rest) = len. of x_2(Rest)
$\bigwedge check(x'_1(Rest), f_2, ..., f_n; f'_2, ..., f'_m)$
// len. of x_1(Rest)>len. of x_2(Rest)
$\bigwedge check(f_2, ..., f_n; x'_2(Rest), f'_2, ..., f'_m)$
// len. of x_1(Rest)<len. of x_2(Rest)

Given that $check(x_1(Rest), f_2, ..., f_n; x_2(Rest), f'_2, ..., f'_m) = disj$, we want to show
$[\![x_1(Rest), f_2, ..., f_n]\!] \bigcap [\![x_2(Rest), f'_2, ..., f'_m]\!] = \varnothing$
By the algorithm, we have that:
$check(f_2, ..., f_n; f'_2, ..., f'_m) = disj$,
$check(x'_1(Rest), f_2, ..., f_n; f'_2, ..., f'_m) = disj$,
$check(f_2, ..., f_n; x'_2(Rest), f'_2, ..., f'_m) = disj$ all hold.
$\Rightarrow [\![f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing$ **A**,
$[\![x'_1(Rest), f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] = \varnothing$ **B**,
$[\![f_2, ..., f_n]\!] \bigcap [\![x'_2(Rest), f'_2, ..., f'_m]\!] = \varnothing$ **C**
where all **A,B,C** statements hold. Note that we express $x_1(Rest)$ as $[\![x_1(Rest)]\!] = [\![x_1[l]]\!] \times [\![x'_1(Rest)]\!] \cup \{\epsilon\}$, $x_2(Rest)$ as $[\![x_2(Rest)]\!] =$

$[\![x_2[m]]\!] \times [\![x'_2(Rest)]\!] \cup \{\epsilon\}$, as by the definition.
We prove by contradiction that
$[\![x_1(Rest), f_2, ..., f_n]\!] \bigcap [\![x_2(Rest), f'_2, ..., f'_m]\!] = \varnothing$.
Suppose there exists some string
$s \in \mathbb{B}^* : s \in ([\![x_1(Rest), f_2, ..., f_n]\!] \bigcap [\![x_2(Rest), f'_2, ..., f'_m]\!])$, meaning that the intersection is not empty. Then, at least one of the following statement sets **1,2,3** must hold (and all included statements in that statement set must all hold), covering all length cases of Rest:
$(s = x[l] \cdot s', s' \in [\![f_2, ..., f_n]\!], s' \in [\![f'_2, ..., f'_m]\!])$ **1**, when lengths of $x_1(Rest), x_2(Rest)$ are the same
$(s = x[l] \cdot s', s' \in [\![x'_1(Rest), f_2, ..., f_n]\!], s' \in [\![f'_2, ..., f'_m]\!])$ **2**, when length of $x_1(Rest) >$ length of $x_2(Rest)$
$(s = x[m] \cdot s', s' \in [\![f_2, ..., f_n]\!], s' \in [\![x'_2(Rest), f'_2, ..., f'_m]\!])$ **3**, when length of $x_2(Rest) >$ length of $x_1(Rest)$
From statement **1**, $[\![f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] \neq \varnothing$, which contradicts statement **A** which holds. Thus, **1** does not hold.
From statement **2**, $[\![x'_1(Rest), f_2, ..., f_n]\!] \bigcap [\![f'_2, ..., f'_m]\!] \neq \varnothing$, which contradicts statement **B** which holds. Thus, **2** does not hold.
From statement **3**, $[\![f_2, ..., f_n]\!] \bigcap [\![x'_2(Rest), f'_2, ..., f'_m]\!] \neq \varnothing$, which contradicts statement **C** which holds. Thus, **3** does not hold.
Since none of **1,2,3** hold, $[\![x_1(Rest), f_2, ..., f_n]\!] \bigcap [\![x_2(Rest), f'_2, ..., f'_m]\!] = \varnothing$.

The presented algorithm is sound, because it does not let errors propagate when handling comparisons, and returns a correct result. Such an error would be a format comparison result of **disj**, when one nested comparison would have resulted to **nondisj**.

The algorithm handles correctly all c, x[l], x[[l]]] fields, because it performs comparisons on byte values. However, we do not treat x(Rest) in a similar manner, since we approximate its behavior. Thus, we cannot claim that our algorithm is complete as well. We also see from the algorithm that a comparison of two formats of unequal lengths will result in the **disj** result.

CHAPTER 4

Concrete Syntax Description

We begin here to describe the elements we use in our language using concrete syntax, i.e. writing down how expressions look like. The concept of declaration is central to our language, since correct input is only input that conforms to the declaration of a message, function, agent or element. The abstract syntax expressions that appear in the sections below are the Haskell data constructors we used for storing the declarations. The implementation stores and fetches all the aforementioned types of declarations in a binary tree, using their identifier as the tree node's value. We use a binary tree for more efficient search times.

4.1 Agent Declarations

We use agent identities, always declared in capital letters. An example of agent declaration is the following one, beginning with the keyword *agent*:

- *agent : SA* An agent declaration, named SA.

The abstract syntax used to store this declaration is *AgentDecl String*.

4.2 Field Declarations

By field declarations we mean the arrays, vectors and varying length field declarations that we define in our language. Depending on the provided values, we also accompany the declaration with the appropriate keyword *const* or *var*, to indicate a field that will have constant or variable values respectively. Below, we show how a syntactically correct field declaration in our language looks like:

- *const fixed_size m : 3* A one byte constant field named m, with the value 3
- *var fixed_size w : 3 | 4* A one byte variable field named w, with two possible values
- *var fixed_size j : ?* A one byte variable field named j, with variable value (could be 0-255)

We refer to fields declared with only one constant value as constants. Variable fields declared with a limited set of values (separated with |) as constrained variables, and variable fields declared with the ? symbol as unconstrained variables.

We include an optional type declaration in each field declaration. The possible basic field types are nonce, key, agent, tag, var or a user-chosen string to represent a custom type. If a type is not given in the field declaration, then the tag type is passed to the relevant data constructor for constant fields, and the var type is assigned to elements with variable values. This behavior happens for all non-type-defined field declarations. More field declarations follow below:

- *const agent fixed_size z[2] : 3,4* A fixed size array of 2 elements named z. It's values are 3,4 per cell
- *var agent fixed_size j[2] : 3,3 | 4,7* A fixed size array of 2 elements named j. Possible values are 3,3 or 4,7
- *var "myType" fixed_size f[2] : ?* A fixed size array of 2 elements named f, with variable values (0 to 256^2-1) and the string myType as field type

The strings that follow the keywords const and var are the strings that define the field types, in these cases the types are agent and customType, which is a user-defined string. For both arrays and one-byte elements, the data constructor used is *ElemDecl declNames :: String, value :: FieldVal* .

The vector declarations are seen below. As reference, we used the vector definitions found in [tls, ch 4.3], where vectors are treated as fields composed of a vector length field, and the actual vector contents. The number between the brackets indicates the number of bytes used as the vector's length field, and the numbers between < .. > the minimum and maximum number of bytes that the vector's contents hold.

- *const agent var_size ac[[1]]<2..2> : 5,6* A constant value vector of type agent, with 2 bytes of content, 1 of length
- *var agent var_size aa[[1]]<0..2> : 4 | 5,6* A variable value vector with min(0)-max(2) content size, with possible values 4 or 5,6
- *var agent var_size ab[[1]]<0..2> : ?* A variable value vector, with possible content lengths of 0 to 2, and of undefined value

Although it does not make much sense to declare constant vector contents, we included it for reasons of language completeness. The data constructor used to store the vector declarations is
VectDecl declNames :: String, min_len :: Int, max_len :: Int, vect_length_field :: FieldVal, vec_value :: FieldVal .

Finally, the variable length fields follow:

- *const agent var_size first(<1..1>) : 2* A constant value of type agent variable length field. Total length is 1 (min & max)
- *var var_size second(<0..3>) : empty | 2 | 1,2,3* A variable length field, ranging from 0 to 3 bytes. Possible values are none, 2 or 1,2,3
- *var var_size third(<1..3>) : ?* A variable length field, ranging from 1 to 3 bytes, and of undefined value

Variable length fields can be used to represent padding fields, which are encountered in some protocol messages. Again, constant valued variable length fields are syntactic sugar, and in essence they represent array fields. The data constructor used for their storage is
VarLenDecl declNames :: String, min_len :: Int, max_len :: Int, var_len_value :: FieldVal

We will use the notion of format when referring to a sequence of fields, which is done frequently in the following sections. We can sum up the field syntax in Figure [?]

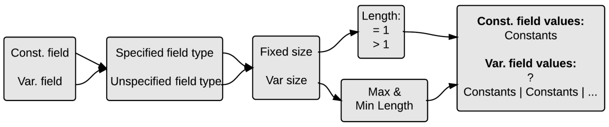

Figure 4.1: All possible field declaration cases

4.3 Format Function Declarations

We use message function declarations, whose use is to represent message fields. The incentive behind their use is the simplification of some formats, and also to reduce their declaration size. Functions are in essence macros, because they map their input according to the function's body declaration. A message function would be declared with the keyword *func*, an identifier used as the function's name, a list of capital arguments of type *field* or *agent*, and the function's body. An example follows:
func func1(field M, field K, agent L) : { M, K }sign(L)
In this case, the function named func1 accepts two field arguments, and one agent argument that is used in the signature key, denoted as sign. A function call for func1 would be *func1(m, j, SA)*, where fields m, j and agent SA should be the identifiers of valid definitions. Declaring and calling functions with inappropriate arguments results in error generation and termination of the program. An example of erroneous declaration in func1 would be the use of the parameter M in the place of L (resulting in sign(M)), where M is declared as a field parameter. A wrong function call would be func1(m,j,j). In order to store the function declaration, we use the data constructor
FuncDecl { declNames :: String, vars :: [TypedParam], func_body :: MsgBody }.
Note that a function's body does not have to include only parameters. Identifiers of defined fields/formats can be used as well.

4.4 Format Declarations

All the declarations above are necessary in order to declare meaningful formats. A format can represent a protocol message, since the sequence and interpretation of each appearing field in a message is clearly defined. An example of a message declaration, using some of the previously discussed declarations would be:
msg msg_1 : m, k, func1(aa, ab, SA)
A format can appear inside another format. Its identifier has to be part of field list of the hosting format. An example is *msg msg_2 : m, msg_1*, where we

include a message inside another message.

To declare formats with one or more possible field sequences, we use the keyword *sub_form*. An example is *sub_form sf_1 : a | b,c | d* , where the identifier *sf_1* can have the field sequences represented by *a*, *b,c* or *d*, respectively. To include a *sub_form* in a format, we simply include its identifier in the format list. E.g. : *msg msg_3 : sf_1* or *sub_form sf_2 : sf_1 | a* . Note that a message cannot have multiple field sequences, only one. Because a *suf_form* format is not considered a message, thus we do not include it in message comparisons.

A protocol is designed to have its messages delivered to the participants that are involved in the protocol's execution, and we can consider that each participant has a role in the protocol run. Although the idea of assigning receiving participants to each message was thought, we decided to not specify any participants for any message in our grammar. The reason is that (although indirectly), we consider that all roles of a protocol/protocols are handled by the same participant, thus we don't need to specify receiving participants. This is why we need to examine the disjointness of all protocol messages, instead of just the message fraction that would correlate to only one role of that protocol like e.g. the client. The drawback of this approach is that it makes it impossible to infer whether a message containing an encrypted field can be decrypted or not, because that would require knowledge of the participant/participants who have the key, as well as the participant's name that receives the message.
Because we had difficulties in defining a semantics function that would distinguish among decryptable and non-decryptable fields, we decided to treat all encrypted fields as ciphertext bytestrings, and to also abandon the idea of message receivers.

Apart from the field declarations that we discussed above, a message can include additional field types. These can be:

- Already defined messages (called by including their identifiers in the message field list).
- Repeated fields. We have three repeated field operators. These operators appear inside a message's body only.
  - [f]: For a field identifier f, found in a message's body, [f] represents the presence of field f, or the absence of it. Equally, we can have the expression [f,g,h] to show more than one optional fields. They all appear, or none appears.
  - f+: For a field identifier f, f+ is equal to infinite repeats of the field f, meaning f+ is equal to f,f,f..... As before, an expression like (f,g)+ is valid, and equal to (f,g)(f,g)....

  - f*: For a field identifier f, f* is equal to [f+]. Expressions like (f,g,h)* are valid. For all these cases, the data constructor used is *RepeatedField { rep_body :: MsgBody, freq :: Reps }*, where the *Reps* value depends on the appearing operator.

- Encrypted fields, defined inside a message's body. An example of that was already seen in the function declaration section, so we will discuss more in detail here. *msg_2 : m,l,{|m,l|}symm(SA,SE)*
  The expression above again is a message declaration, containing an encrypted field. The encrypted content is enclosed within the {|, |} symbols, and the keyword *symm* shows the use of a symmetric encryption key. For asymmetric encryption, we would have the encrypted content contained inside the {, } symbols, and the public/private asymmetric encryption keys would be represented with the keywords *public, sign* respectively. Encrypted fields can be found in function or message declarations only. The data constructor for encrypted fields is *EncryptedField { sub_msg_body :: MsgBody, encr_key :: KeyType }*.

CHAPTER 5

Semantics Function

A semantics function is a function accepting an input from a specific domain, in this case the input is our language, and returns its output value. We represent a semantics function as $[\![$ input $]\!] \rightarrow \mathcal{P}\,(String)$, where in our case, the input is all the possible fields that appear in our grammar, and $\mathcal{P}\,(String)$ is the powerset of String, meaning all the possible combinations of values of strings. We use the $\times$ operator to indicate the concatenation operation on sets of strings: $A \times B = \{a \cdot b | a \in A, b \in B\}$, where $\cdot$ is the string concatenation operator, A, B sets of strings. We define as B the set of possible strings that a one byte long value can have, B^n the set of byte strings of length n. Thus, $B^n = B \times B \times ... \times B$, with B appearing n times. c is a 1 byte long constant value, C a constant of k bytes length, where k $\in$ 1.. ∞. So, C $\equiv$ $(c_1,c_2,...,c_k)$. We represent as S a string or arbitrary length.

$[\![$. $]\!]$:: Field Declaration $\rightarrow B^+$

The declarations *const fixed_size name* = *C*, *?*, C_1 | C_2 | ... | C_n are syntactic sugar, and are equivalent to the 1 byte length array declaration, showed above.

- $[\![$ const fixed_size name[n] : c_1,c_2,...,c_n $]\!]$ = { $(c_1,c_2,...,c_n)$ | $c_i \in B$, i $\in$ {1..n} }, c_i is a known value constant of byte length 1.

- $[\![$ var fixed_size name[n] : C_1 | C_2 | ... | C_f $]\!]$ = { C_1, C_2, ... C_f | $C_i \in B^n$, i ∈ {1..f} }, where C_i is a constant.
- $[\![$ var fixed_size name[n] : ? $]\!]$ = B^n, where B^n the set of possible values of an n-length byte string
- $[\![$ const var_size name[[n]]<l..l> : C $]\!]$ = { uv | u = l, u ∈ B^n, v ∈ B^l, C ∈ B^l, v = C }, C is a constant.
- $[\![$ var var_size name[[n]]<l..m> : C_1 | C_2 | ... | C_f $]\!]$ = { uv | uv ∈ {l_1C_1, l_2C_2, ..., l_fC_f}, $C_i \in B^l \bigcup B^{l+1} \bigcup ... \bigcup B^m$, $l_i \in B^n$, i ∈ {1..f} }, the values of l_i are equal to the byte lengths of its corresponding C_i element.
- $[\![$ var var_size name[[n]]<l..m> : ? $]\!]$ = { uv | uv ∈ $\{u_1v_1, u_2v_2, ..., u_{m-l+1}v_{m-l+1}\}, u_i \in B^n, v_i \in B^{i-1+l}, i \in \{1..(m-l+1)\}$ }, the values of the appearing u_i are equal to the byte lengths of v_i.
- $[\![$ const var_size name(<l..l>) : C $]\!]$ = { C | C ∈ B^l }, where C is a constant.
- $[\![$ var var_size name(<l..m>) : C_1 | C_2 ... | C_f $]\!]$ = { u | u ∈ { C_1, C_2, ... C_f }, $C_i \in B^l \bigcup B^{l+1} \bigcup ... \bigcup B^m$, i ∈ {1..f}, f ∈ {1..(m-l+1)} }
- $[\![$ var var_size name(<l..m>) : ? $]\!]$ = { u | u ∈ $B^l \bigcup B^{l+1} \bigcup ... \bigcup B^m$ }

We present the semantics function for the encrypted fields.
$[\![$. $]\!]$ :: Field → B^+

- $[\![$ {|form|}symm(A,B) $]\!]$ = B^+. Form is a sequence of fields and A,B the two agents, who hold the shared encryption key. The encrypted content is the form. The length of the encryption is not known.
- $[\![$ {form}public(A) $]\!]$ = B^+, as defined above.
- $[\![$ {form}sign(A) $]\!]$ = B^+, as defined above.

A function, encountered as a field in a message form and noted as *func_name(arg1, arg2, ... , argn)*, is syntactic sugar for the sequence of fields that the function declaration represents. When encountering a function call, it will always be a field encountered in a message form. Replacing the function call arguments in the function declaration gives us the resulting message form that the function call represents. As an example, assume we have a function declaration *f(agent A, field M) : { M, M }public(A)*, and a possible function call would be *f(SA, m)* with the parameters SA, m defined elsewhere.

A form is a sequence of fields, and we represent a form as form $= field_1$, ... ,$field_n$ where n $\in$ N, and $field_n$ can be one of the possible cases of fields we defined previously. If n=0, then form $= \epsilon$, the empty word. A sub_form is composed of forms, separated by the | symbol. We represent a sub_form as sub_form $= form_1 \mid form_2 \mid ... \mid form_n$, where n $\in$ N$\backslash\{0\}$.

- $[\![\ .\ ]\!]$ :: Form $\rightarrow B^+$
- $[\![\ \epsilon\ ]\!] = \{\ \epsilon\ \}$, where ϵ is the empty word.
- $[\![$ form $]\!] = \{\ \epsilon\ \} \bigcup [\![\ field_1\ ]\!] \times [\![\ field_2\ ]\!] \times ... \times [\![\ field_n\ ]\!]$
- $[\![$ [form] $]\!] = [\![$ form $]\!] \bigcup \{\ \epsilon\ \}$.
- $[\![\ form^+\ ]\!] = [\![$ form $]\!] \bigcup [\![$ form, form $]\!] \bigcup$ .... Equally: $\bigcup\limits_{i=1}^{\infty} [\![\ form^i\ ]\!]$.
- $[\![\ form^*\ ]\!] = \{\ \epsilon\ \} \bigcup [\![\ form^+\ ]\!]$
- $[\![\ sub\_form\ ]\!] = [\![form_1]\!] \bigcup [\![form_2]\!] \bigcup ... \bigcup [\![form_n]\!]$

Chapter 6

Unification

Very generally speaking, unification tries to identify two symbolic expressions by replacing certain sub-expressions (variables) by other expressions [7]. We consider terms that are built from function symbols (of n-arity) and variables. The arity of a function symbol can also be zero. The set of variables is denoted with $\mathcal{V}$, and the set of function symbols with $\mathcal{F}$. Syntactic unification works by replacing the variables found in expressions with terms, such that the resulting expressions will be syntactically equal [7]. Variables that represent functions constitute higher-order variables. A solution to the first-order unification problem, noted as σ, is called substitution. σ is applied suffix to a term. Solving equations that contain higher-order variables is called higher-order unification.

Throughout this work, we are interested only in first-order unification. We view the constant byte values of fields as function symbols of zero arity $\in \mathcal{F}$, and fields with free variable values as variables $\in \mathcal{V}$. We are not interested in finding the actual substitution among two terms. Instead, we want to know if two field values are unifiable. If so, the fields they represent are confusable, otherwise they are disjoint. For simplicity, we refer to free/constrained variable fields as variables, and constants as function symbols of zero arity throughout the rest of this section only.

Since variables represent arbitrary values, confusion among a variable/constant pair of terms or variable/variable pair of terms is possible, if there exists a substitution. As mentioned, function symbols cannot be substituted in first-order unification. We only examine first-order unification in our implementation.

As an example, consider the equation var[32] = const_val[32], with the first term being a free variable field, and the second term being a constant value field, both 32 bytes long. A substitution that would make both terms equal would be $\sigma = \{$ var[0] $\rightarrow$ const_val[0], ... , var[31] $\rightarrow$ const_val[31] $\}$, denoting a byte-to-byte substitution, or for simplicity, $\sigma = \{$ var[32] $\rightarrow$ const_val[32] $\}$ for the entire terms. The aforementioned substitution would result in the equality relation : var[32]σ = const_val[32]σ.

Unification conditions: One necessary condition for messages to be potentially confusable is to be of equal length. This is necessary in order to have a meaningful substitution function. Also from the point of view of the OS/protocol execution, a message with a length longer than the expected (trailing bytes) could be accepted as valid message, but could also be rejected as malformed. Since this behavior is protocol implementation dependent and there might be variances in it, we demand equal lengths in each examined message pair, that also make the substitution function meaningful.

Chapter 7

Processing Format Fields

From the concrete syntax we saw that each field declaration includes value assignments as well. This is necessary in order to have meaningful value comparisons in the checker. This section discusses the Haskell data types used for representing the values a field can have.

As mentioned before, the fields that appear in a message are the ones described in the Field and Message declaration sections. For the corresponding data constructors of each field that appears in a message, we refer the interested reader to the *Field* data type declarations located in the grammar file.

The input to the checker is a set of declarations, which must include some message declarations as well, if we wish to have message comparisons. However, since the comparison works on byte values, we have to resolve a declaration to its byte values. We discuss initially how byte values are represented.

7.1 Field Value Representation

Byte values are represented by the data type FieldVal, as seen below.

```
data FieldVal = Bytes Size Value -- For constant size fields
              | VectFieldVal FieldVal FieldVal -- For vectors
              | RepFieldVal [FieldVal] Reps Int -- For *,+,[] fields
              | EncrFieldVal [FieldVal] KeyDef
              deriving (Show, Read, Eq)
```

The first data constructor is used to represent the value that a field declaration can have. The *Bytes* data constructor takes two additional arguments which are *Size* and *Value*. We use three different field sizes: fixed size, representing elements like arrays, variable size to represent elements like vectors and variable length fields, and the field size *Rest*, to represent the size of encrypted fields. The first two sizes are self-explanatory.

```
data Size = FixedSize Int
          | VarSize Int Int  -- Min and Max bytes in the vector
          | Rest
          deriving (Show, Read, Eq)
```

Rest is a very generic way to represent encryption lengths, and does not give any actual length values. We should point out that *Rest* is used because, as mentioned before, in the protocols we examined the encrypted field is always the last. Each message found in TLS and IKE containing an encrypted field has a message length field, whose value is the total message length value. Using this message length value, it is possible to determine the length of the encrypted field as well. So *Rest* is used to simplify encrypted field descriptions. This has a certain drawback though: For a protocol that uses two or more encryptions in the same message, to use *Rest* as the encryption length of each encrypted field is not appropriate. An alternative to this approach would be to compute the length of the encrypted content (say *len*), and represent it as a *CipherText* valued field, with a length of *len to (len+2048)*, which is the maximum size increase an encrypted content should have in TLS [4, p. 21]. However, the drawback of this approach is that it is not always feasible to compute the exact size of the encrypted content. For example, when the encrypted content contains an infinitely repeated field/field list (noted with the * operator) we cannot know the number of times the field will appear, which means we should again try all possible cases of field appearances e.c.t. Since this increases the computational burden substantially, we decided to just use *Rest*, and leave the appearance of more than one encrypted fields in a message as an unsupported feature for now.

To represent the actual byte values, we use the *Value* data type, as seen below:

```
data Value = Known [[Int]] FieldType
           | VarValue FieldType
           | CipherText
           deriving (Show, Read, Eq)
```

The *Known* data constructor stores the byte values we pass to a field. For example, the field declaration *const agent fixed_size g[2] = 3,4* would result in the field g having a *Value* instance of *Known [[3,4]] AgentType*. A vector field declaration such as *var var_size j[[1]]<2..4> : 1,2 | 4,5,6* would result in a vector content value of *Known [[1,2],[4,5,6]] VarType*. The *VarValue* represents any possible value the field can have, and is used when a field declaration is assigned the *?* symbol. As mentioned above, the *CipherText* value is used for encrypted fields.

Since the *Bytes* data constructor has been examined, we will focus on the remaining *FieldVal* data constructors. The vector's values are represented by *VectFieldVal FieldVal FieldVal*, where the value of the first *FieldVal* field represents the vector's length field, and the second *FieldVal* represents the vector's actual contents.

Repeated fields are represented by the data constructor *RepFieldVal [FieldVal] Reps Int*, where *[FieldVal]* represents the possible values the repeated field/field list can have. *Reps* represents the number of times a field/field list appears, depending on the choice among the [],+,* operators. The *Int* field is instantiated with zero, and is used afterwards during message comparison, to count the number of field/field list repeats.

An encrypted field's value is represented by *EncrFieldVal [FieldVal] KeyDef*, where *KeyDef* represents the used key. We already mentioned that an encrypted field has the *CipherText* value, so the *EncrFieldVal* might seem redundant or a contradiction. However, we believe that it makes the implementation more sound, and might be fully used when messages with multiple encrypted fields are fully supported in the future.

Chapter 8

A Quick Tour On The Implementation

This section discusses the format comparison part of the implementation. The check algorithm has been presented already, so in this section we present with more detail how the implementation performs the algorithm's steps. Remember that the implementation does not follow the algorithm's behavior exactly.

When examining the disjointness of a message set, we follow a sequence of steps, from parsing to comparison. For a message set consisting of $msg_1, ..., msg_n$, we perform the following steps for all msg_i belonging to the set. n must be $\geq$2 if we are to perform comparisons (step 4).

1. Store the message declaration of all msg_i, using the data constructor *MsgDecl { declNames :: String, body :: [Field] }*. We also retrieve the sub-message patterns (SMP) found in the encryptions (if present) of msg_i. The encrypted payload of an encrypted field is a SMP. The list *[Field]* holds the field identifiers that appear in the message's body.
2. For each element of the aforementioned *[Field]* list, fetch its corresponding declaration (in order to retrieve the value/values of that field).
3. Construct the possible values set of msg_i. Considering the possible values of each field in the message, construct the set of possible values of the

message, and use the *MsgValues String [[FieldVal]]* data constructor to store it. The list of lists *[[FieldVal]]* holds the set of possible values of the message. This step is equal to computing $[\![msg_i]\!]$.

4. Examine the disjointness of two messages by comparing their value sets. All elements of the first value set are compared with all elements of the second value set. If a value exists in both sets, the comparison terminates, and the message pair is declared non-disjoint. The comparison tries to determine the value of the expression $[\![msg_1]\!] \cap [\![msg_2]\!]$, where msg_1, msg_2 is the examined pair.

```
compare_two_msgs :: Msg -> Msg -> (IO(), [(Bool, [String])])
compare_two_msgs (MsgValues name_1 val_list_1)
                 (MsgValues name_2 val_list_2) =
  -- each element of val_list_1 is combined with each element of val_list_2,
  -- and fed to the lambda function
  (putStr( "*** " ++ name_1 ++ " compared with " ++ name_2 ++ " *** "),
     nub (fmap (\s -> runWriter s) $
             (\msg_1 msg_2 -> compare_msg_values msg_1 msg_2)
              <$> val_list_1 <*> val_list_2))
```

Figure 8.1: The Haskell code that compares a pair of message values. Each message's values are contained in their respective val_list_1,2 list.

Step 4 is performed on all message pairs resulting from the union of the $msg_1, ..., msg_n$ set, and the resulting SMP set. Thus, we can reach a result on the disjointness of the message set $msg_1, ..., msg_n$.

The Haskell code for comparing the value sets of two messages (step 4) is seen in Figure 8.1. We use the applicative functor approach (operator <*>) to compare each possible pair of *val_list_1,val_list_2*, and feed that pair to the *compare_msgs* function, that performs the value pair comparison. When comparing a set of messages, we also include the resulting SMPs from the message set in the comparison. Thus, we perform all possible comparison pairs between messages and SMPs.

8.1 A Message's Possible Values

Before performing message comparisons, we perform a-renaming on the field values. When a-renaming a variable, we assign it a different value on each renaming. In our implementation however, we do not reassign any values, but only change the field identifiers accordingly, adding the {n} string next to the identifier, with n equal to 1,2.... In our implementation, a-renaming has no practical importance, and is done just as a formality. We a-rename only fields of variable type, or names of agents that appear for example in encryption keys. Constant fields of type *Tagtype* are left untouched.

If a field has more than one possible values, a message containing such a field will also have more than one possible values. For each field that can have more than one *Known* values, we have to consider all of these values when computing the message's value. Cases of such field instances were mentioned above, and we will discuss a few examples here, including the implementation's output.

- An array declaration like *var agent fixed_size j[2] : 3,3 | 4,7* will result in the declaration: *ElemDecl {declNames = "j", value = Bytes (FixedSize 2) (Known [[3,3],[4,7]] AgentType)}*. Since the *Known* list contains two elements, these are the two possible values of this array. The message declaration *msg f1 : j* will have the value list:

  ```
  MsgValues "f1" [[Bytes (FixedSize 2) (Known [[3,3]] AgentType)],
                  [Bytes (FixedSize 2) (Known [[4,7]] AgentType)]]
  ```

  We see that the list contains two elements of type *Known*, with the first containing the values 3,3 and the other the values 4,7.

- We consider the vector declaration *var agent var_size ab[[1]]<0..2> : ?* and the message declaration *msg f2 : ab* . The vector *ab* can have three possible element cases, from zero bytes as vector content, to two bytes vector content. The byte values are undefined, and represented using *VarValue*. A vector's length field always shows the byte length that the vector's content has. In each case, the vector's length changes value, always according to the contents. The implementation choice of creating all possible vector length cases, and testing all of them, resulted as a solution to the difficulty of statically processing a vector. By considering all cases of vector lengths we end up having deterministic vector content for each case, that is, we know its length and value. This makes field and message value comparison feasible. In contrast, it turned out to be very difficult to correctly handle vectors without using this approach. So, although we might require more computations this way, the comparison is feasible.

  For the aforementioned vector, the possible values output of the checker is seen in Figure 8.2: We see the three cases of vector content, as well as the values 0,1,2 that the vector's length field has in each case. Equally, the *FixedSize* data type of the vector's content has the values 0,1,2 to represent the vector content length.

```
MsgValues "f2" [[VectFieldVal (Bytes (FixedSize 1) (Known [[0]] AgentType))
                              (Bytes (FixedSize 0) (VarValue AgentType))],
                [VectFieldVal (Bytes (FixedSize 1) (Known [[1]] AgentType))
                              (Bytes (FixedSize 1) (VarValue AgentType))],
                [VectFieldVal (Bytes (FixedSize 1) (Known [[2]] AgentType))
                              (Bytes (FixedSize 2) (VarValue AgentType))]]
```

Figure 8.2: The possible values of the vector declaration *var agent var_size ab[[1]]<0..2> : ?*, as outputted by the implementation.

Figure 8.3 shows the 3rd step (value set construction) of a message. Note that each field now holds only one value. Expressed formally, a field f_i has the value set $[\![f_i[\![$, and a n-field format $F = f_1, f_2, ..., f_n$ has as possible values the set $S = [\![f_1]\!] \times [\![f_2]\!] \times ... \times [\![f_n]\!]$.

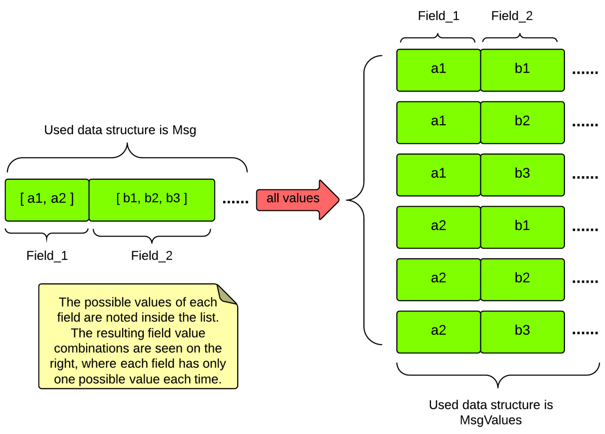
Used data structure is Msg
[a1, a2]
[b1, b2, b3]
......
Field_1
Field_2
all values
The possible values of each field are noted inside the list. The resulting field value combinations are seen on the right, where each field has only one possible value each time.
Field_1
Field_2
a1
b1
a1
b2
a1
b3
a2
b1
a2
b2
a2
b3
Used data structure is MsgValues

Figure 8.3: All possible message values

The handling of repeated and subform fields is slightly different though. We compute the possible values of these fields at step 4, which is during the actual message value comparison. For all other fields, this happens at step 3. This means that the set of values of repeated fields during the 3rd step is composed of only one element. The reason is because this is a correct way to compare repeated fields. For example, consider a message *msg* $f_1 : a, (b)^*$, where *a* is a constant value, but *b* can have the values *b1* and *b2*, where all *a*,*b1*,*b2* can be (but not necessarily are) more than one byte values. With every repeat of *b*, we cannot be sure which of the *b1* or *b2* values will appear, since for example *a*,*b1*,*b2*,*b1*,*b1*,*b2*,... is a valid run, and *b*'s value changes. So instead of creating the infinitely many possible values of f_1 and store them in the *MsgValues* data type, we compare all repeated fields like b during the message value comparison (step 4). We explain how this works in a future section.

Subform fields could be handled in step 3 as well. However, we preferred to not increase the size of the possible message value set further. Instead, we consider their field value set in step 4.

Note that due to the different treatment of repeated and subform fields, we break the formal representation $S = [\![f_1]\!] \times [\![f_2]\!] \times ... \times [\![f_n]\!]$ we use for a value set in step 3. To be concise with the formal notation, we would have to either declare a new semantic function for this intermediate state of repeated fields and subforms, or not use the above formal notation.

8.2 Performing A Value Pair Comparison

Value comparison among two message value sets (step 4) works by comparing each message value pair byte by byte. Consider that messages msg_1, msg_2 result in the message value sets $valset_1, valset_2$ each, with $valset_1 = \{val_1, ..., val_n\}$, $valset_2 = \{val'_1, ..., val'_m\}$. We compare all val_i, val'_j pairs, where $i \in 1..n, j \in 1..m$. The first field value of both val_i, val'_j is compared. When depleted on either val_i or val'_j, the next field value that follows it is fetched e.c.t. The comparison continues to run for as long as it finds the examined field values confusable. If however we reach a point in execution where the examined field value pair of val_i, val'_j is not confusable, we conclude that val_i, val'_j are not confusable as well. Figure 8.4 shows a simple message value comparison, where the compared field values do not have the same lengths. In this case, each of the examined field values has only one possible value (it is not a repeated field), thus is does not resolve itself into more than one values.

The case where a field value val_i resolves further into more field values concerns the repeated fields, that is, fields that appear using the [],*,+ operators in a message. The rationale was explained in section 8.1, and Figure 8.5 depicts this functionality schematically. Same behavior is exhibited by the subform fields. We decided to set a limit on the number of repetitions an infinitely repeated field (noted with +/*) can have. This sets an upper limit to the resulting branching, because on some occasions that can become quite a lot. A difficult case for example is encountered in IKE, where we have a message containing three nested infinitely repeated fields. The variable controlling the allowed repeats of an infinite field is *max_ spawns_ allowed*.

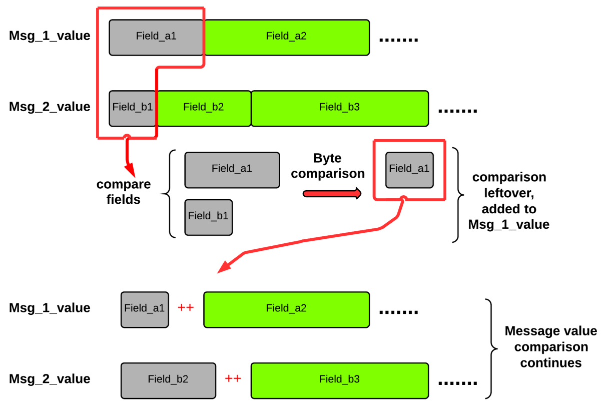

Figure 8.4: Message value pair comparison, with one possible value per element.

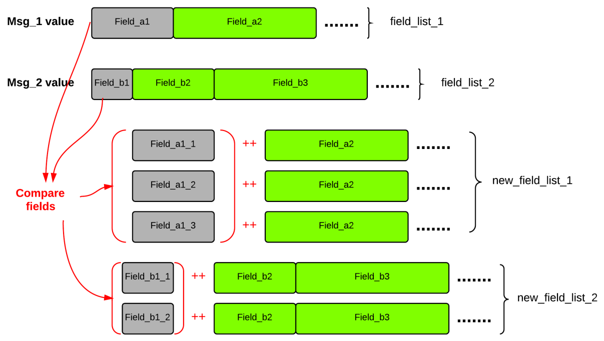

Figure 8.5: A message value comparison, where both compared field values resolve into more than one possible field values. After the creation of the new_field_list_1,2 lists, all possible pairs constructed from the elements of new_field_list_1,2 lists are compared. We implement this functionality in our implementation with
$(\backslash x\ y -> compare\_msg\_values\ x\ y)$
$< \$ > new\_field\_list\_1 < * > new\_field\_list\_2$. Note that the above process applies to all fields, whether they have one or more possible values. However, we explicitly show in this figure the multiple values case.

The function *compare_msg_values* performs the comparison among a val_i, val'_j pair. The two arguments are lists of *FieldVal*, and hold the field values of

val_i, val'_j respectively. The code fragment that performs this operation is seen in Figure 8.6.

```
compare_msg_values :: [FieldVal] -> [FieldVal] -> Writer [String] Bool
compare_msg_values (head_1:field_list_1) (head_2:field_list_2) =
  -- Adds what's left from the last field comparison (head_1,2) to the rest
  -- of the field_list of each message.
  do
  let (res, leftover_list_1, leftover_list_2) = compare_fields head_1 head_2
      new_field_list_1 = (++) <$> (filter_leftover <$> leftover_list_1) <*> [field_list_1]
      new_field_list_2 = (++) <$> (filter_leftover <$> leftover_list_2) <*> [field_list_2]

  if res == True && new_field_list_2 /= [] then -- comparison not over yet
     do
     let writer_logs = (\x y -> compare_msg_values x y)
                       <$> new_field_list_1 <*> new_field_list_2
     -- ORing the result of the above compare_msg_values run
     ored_res = or $ fmap (\s -> fst $ runWriter s) writer_logs
     strings = concat $ fmap ((\(val,str)-> if val==True then str else [])
                              . (\s -> runWriter s)) writer_logs
     tell strings
     return ored_res
  else if res == True && new_field_list_2 == [] then
     do
     tell ["Same, depleted msg_2"]
     return True
  else do
     tell []
     return res -- last byte comparison returned False
```

Figure 8.6: The Haskell code that performs the field value comparison. Its input is a pair of message values.

The first field value of each *FieldVal* list is compared, namely *head_1, head_2*, using the *compare_fields* function. The resulting possible values of *head_1, head_2* are stored in *leftover_list_1, leftover_list_2*. Note that *leftover_list* will contain one element only if its corresponding *head* field value cannot resolve further (as in Figure 8.4), or more than one elements if the field resolves into more than one possible values (as seen in Figure 8.5). The code that constructs the new set of the remaining possible values of val_i and val_j is seen below:

```
new_field_list_1 = (++) <$> (filter_leftover <$> leftover_list_1) <*> [field_list_1]
new_field_list_2 = (++) <$> (filter_leftover <$> leftover_list_2) <*> [field_list_2]
```

Note that the remaining field values of both val_i and val_j are stored in *field_list_1,2* and haven't been examined yet. When the above code segment runs, all remaining unexamined field values of val_i and val_j are stored in their corresponding *new_field_list_1,2* list. If the comparison has not ended yet, they are used as input arguments of the recursive call of *compare_msg_values*, until the comparison terminates. The corresponding code segment is seen below:

```
(\x y -> compare_msg_values x y) <$> new_field_list_1 <*> new_field_list_2
```

8.2.1 Type Handling

When a message value pair is compared, we also retrieve the basic or composed types that they have. Our implementation outputs the types of both messages, but gives no additional info, or perform any comparison on types. There can be cases of message pairs that are confusable but also have different types. That means there exists a unifier among these two messages, but that unifier will not preserve field types. Instead, it treats fields as byte values, disregarding types.

An example of the above case would be the comparison of the following two formats:

msg f_1 : A, B, Kab , where *Kab* is a key, *A,B* are constants.
msg f_2 : A, B, NA, NB, where *NA, NB* are nonces.

Say that both *f_1, f_2* have the same total byte lengths. For these messages this means that a substitution exists, however since these two formats have different types, that substitution will not respect types. If however their types were the same, then we would have an honest substitution (one that respects types).

8.3 Field Comparisons in the Implementation

In message comparison, we have fields that can be of constant or variable value. We also consider the encrypted fields as variable fields and note them as ciphertext.

When comparing a pair of variable fields, or a variable with a constant field, we are not interested in finding the set of all possible substitutions the value pair might have. Instead, we examine unifiability among fields, i.e. whether there can exist a unifier. If a unifier exists, then the fields are confusable. Comparison of constant field pairs is more straightforward, because we compare their byte values directly. When comparing fields, we perform a byte-to-byte comparison, as seen in Figure 8.4: we compare one byte at a time until the shorter field depletes, or until we discover they are disjoint.

In the discussed comparisons below, we discuss comparing bytes that are both one byte long. We abstract from the lengths of the compared fields, and because we compare fields byte-by-byte. As mentioned, we generate all the possible values of a message (say $val_1, ..., val_n$), in order to perform the disjointness check. As depicted in Figure 8.3, a constrained variable will have only one of its values, each time it appears inside any of the $val_1, ..., val_n$ values.

- **Constant & Variable**:
  When comparing constants with constrained variables (both 1 byte long), we compare known byte values. If they are unequal, the examined message value pair is unequal. Otherwise, the comparison continues.

  Among an unconstrained variable var_t and a constant const_t, there always exists the substitution $\sigma =\{$ var_t $\rightarrow$ const_t $\}$. Similarly, there always exists a substitution among a 1-byte long constrained variable and a 1-byte long free variable.

- **Ciphertexts** :
  The ciphertext fields are treated as free variables. That means they are confusable with another free/constrained variable or constant. We do not distinguish ciphetexts that occur from symmetric and asymmetric cryptography.

  Consider the comparison among a ciphertext and a constant. Having a unifier among a constant and a ciphertext would mean that the content we encrypt is chosen in a particular way, so that its resulting ciphertext matches byte-by-byte the constant field. This scenario is unlikely to happen in reality, however the intersection between a constant and a ciphertext is not empty. Since we cannot exclude that ciphertexts and constants can be unified, we consider the worst case scenario. Thus, we treat ciphertexts and constants as confusable.

  Because we treat ciphertexts as free variables, we also consider both free and constrained variable fields and a ciphertext field as unifiable.

  Finally, we treat two compared ciphertexts as confusable. For two ciphertexts (resulting from different encrypted contents) to have the exact same byte values, we would have to encrypt the same content, and use the same algorithm and keys. If we were to think in terms of protocol execution, an attempt to decrypt a random/wrong bytestream (confused as ciphertext) would fail, and the protocol's execution would not continue. However, we do not approach this comparison in terms of protocol execution, but from a static analysis point of view. Also, by considering two ciphertexts as confusable, we are consistent with our view of ciphertexts as variables.

The implementation does not examine comparisons involving ciphertexts with the byte-by-byte approach. As mentioned in section 7.1, we have no length field for a ciphertext. Thus, we compare ciphertexts on the field level, not on the byte value level. For example, consider the comparison of a 5 byte long field value, consisting of free variables, and a ciphertext being compared. The implementation would discard the variable field, and would continue the comparison by comparing the next field that follows the variable, again with the ciphertext. This implementation approach could be problematic if we were to encounter encrypted fields in the middle of a message. When building the implementation, we had in mind that encrypted fields are always the last field of a message, since that was the case for our examined protocol set. However, if we were to make a more wide message case study, this approach would be problematic.

Although simplistic, the aforementioned treatment of ciphertexts does not reduce the correctness of the disjointness check we perform. The encrypted contents are examined through the SMP message sets, so no loss of scrutiny occurs in the examined message set. Also, the definition of the semantics function for an encrypted field turned out to be very difficult, if we were to consider more properties of ciphertexts like decryptability.

8.4 A Short Example

In this section we will present a small set of formats depicted in Figure 8.7, and the output produced by the checker when comparing them. This example uses a section of the implemented work in terms of appearing concrete syntax and functionality of the checker, but not all.

The checker's output is seen in Figure 8.8. The appearing SMPs are the formats that constitute the encrypted content. They are denoted with the prefix smp_, followed by the message identifier from which we extracted them. Note that False represents the distinguishability of a format pair, True means that two formats are non-disjoint (confusable). Some additional information is included in the boolean result, depending on the comparison case. The intention is to provide more information on a particular comparison result.

As seen from the output, the SMPs are problematic because they are not distinguishable enough. Also, the same happens with messages f1-f2,f1-f3 because of

```
agent : A
agent : B
agent : C

const tag fixed_size tag_1 : 1
const tag fixed_size tag_2 : 2
const tag fixed_size keytag : 7
var nonce fixed_size nonce_a[64] : ?
var nonce fixed_size nonce_b[64] : ?

msg f1 : { nonce_a, keytag }public(B)
msg f2 : tag_2, { keytag, nonce_a }public(A)
msg f3 : tag_1, { nonce_a, keytag }public(C)
```

Figure 8.7: A small problematic format set.

the lack of some starting tag on f1. f1 is a ciphertext field, which is treated as a variable. As mentioned, when comparing a pair of ciphertexts, we consider them to be confusable. Along with the comparison result, we also output the type of each format. For formats with more than one fields, it will be a composed type.

In order to fix the problematic formats, we introduce additional tags in the formats, aiming to make them distinguishable. The changes are seen in Figure 8.9

After the introduction of additional tags, the checker returns all formats to be distinguishable.

```
*** f1 compared with f2 *** [(True,["Same, depleted msg_2"])]
    Appearing types: "crypt(nonce,tag)" & "tag,crypt(tag,nonce)"
*** f1 compared with f3 *** [(True,["Same, depleted msg_2"])]
    Appearing types: "crypt(nonce,tag)" & "tag,crypt(nonce,tag)"
*** f1 compared with smp_f1 *** [(True,["Same, depleted msg_2"])]
    Appearing types: "crypt(nonce,tag)" & "nonce,tag"
*** f1 compared with smp_f2 *** [(True,["Same, depleted msg_2"])]
    Appearing types: "crypt(nonce,tag)" & "tag,nonce"
*** f1 compared with smp_f3 *** [(True,["Same, depleted msg_2"])]
    Appearing types: "crypt(nonce,tag)" & "nonce,tag"
*** f2 compared with f3 *** [(False,[])]
    Appearing types: "tag,crypt(tag,nonce)" & "tag,crypt(nonce,tag)"
*** f2 compared with smp_f1 *** [(True,["Same, depleted msg_2"])]
    Appearing types: "tag,crypt(tag,nonce)" & "nonce,tag"
*** f2 compared with smp_f2 *** [(False,[])]
    Appearing types: "tag,crypt(tag,nonce)" & "tag,nonce"
*** f2 compared with smp_f3 *** [(True,["Same, depleted msg_2"])]
    Appearing types: "tag,crypt(tag,nonce)" & "nonce,tag"
*** f3 compared with smp_f1 *** [(True,["Same, depleted msg_2"])]
    Appearing types: "tag,crypt(nonce,tag)" & "nonce,tag"
*** f3 compared with smp_f2 *** [(False,[])]
    Appearing types: "tag,crypt(nonce,tag)" & "tag,nonce"
*** f3 compared with smp_f3 *** [(True,["Same, depleted msg_2"])]
    Appearing types: "tag,crypt(nonce,tag)" & "nonce,tag"
*** smp_f1 compared with smp_f2 *** [(True,["Same lengths"])]
    Appearing types: "nonce,tag" & "tag,nonce"
*** smp_f1 compared with smp_f3 *** [(True,["Same lengths"])]
    Appearing types: "nonce,tag" & "nonce,tag"
*** smp_f2 compared with smp_f3 *** [(True,["Same lengths"])]
    Appearing types: "tag,nonce" & "nonce,tag"
```

Figure 8.8: The checker's output, after examining the problematic format set.

```
agent : A
agent : B
agent : C

const tag fixed_size tag_1 : 1
const tag fixed_size tag_2 : 2
const tag fixed_size tag_3 : 3
const tag fixed_size inner_tag_1 : 4
const tag fixed_size inner_tag_2 : 5
const tag fixed_size inner_tag_3 : 6
const tag fixed_size keytag : 7
var nonce fixed_size nonce_a[64] : ?
var nonce fixed_size nonce_b[64] : ?

msg f1 : tag_1, { inner_tag_1, nonce_a, keytag }public(B)
msg f2 : tag_2, { inner_tag_2, keytag, nonce_a }public(A)
msg f3 : tag_3, { inner_tag_3, nonce_a, keytag }public(C)
```

Figure 8.9: Corrected format set. Now there are no confusability reports from the checker.

Chapter 9

XML Implementation

Due to the lack of time, some features concerning support for XML messages were left unfinished. We discuss the current status of our XML support, as well as the needed work to have it fully covered.

9.1 XML Concrete Syntax

XML (eXtensible Markup Language) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. By definition, an XML document is a string of characters. The character encoding used in XML messages are the UTF8 and UTF16 encodings. Proper XML syntax rules are explained in [3], where the concepts of XML element, attribute and content are discussed through a set of examples.

Our grammar provides support for representing simple and complex XML elements. We briefly mention the work done on XML, by presenting some examples below.

- *string se : "mystring"*
  A string declaration, with the value *mystring*, and the identifier *se* as

the string's name. This identifier will be used when passing this string declaration as an argument.

- $xml\_elem\ cd1 :< ''Tagname'' > ''cdcont'' < /''Tagname'' >$
A simple XML element declaration, named cd1. The keyword *xml_elem* indicates an XML element declaration. The strings *Tagname, cdcont* are the XML element's tag name and XML element's content, respectively. We enclose them in quote characters "", to indicate they should be interpreted as a string. XML elements declared with the *xml_elem* keyword must contain only strings (strings enclosed in "") as tag names, attributes and content.

- $xml\_elem\ cd3 :< ''Topname''\ ''src'' = ''''attrname'''' >< ''Tname'' > ''cdcont'' < /''Tname'' >< /''Topname'' >$
An XML element declaration, containing a nested XML element (child element) as content. If more than one nested XML elements are used as content, they must be separated by commas. Notice that we use double quotes on attributes, because we want the string "*attrib_name*" to be interpreted as the XML attribute. The above XML element, as well as all our syntax concerning XML, can be viewed as a flattened XML message. Thus, the above syntax is the same as having :

```
<"Topname" "src"=""attrname"">
   <"Tname">"cdcont"</"Tname">
</"Topname">
```

which resembles the appearance of an ordinary XML element.

- $func\ func\_3(string\ S,\ xml\_elem\ E,\ fieldF\ ) : < ''msg'' > E, < S > F < /S >< /''msg'' >$
An XML function declaration. An XML function should be viewed as a parameterized XML element. XML functions work in the same spirit as format functions mentioned in section 4. In essence, they also are macros. We can pass three types of arguments in an XML function: strings, xml elements, and the fields described in 4. As with format functions, variables are written in capital letters. We do not use any quote characters on the variables, because then they would be interpreted as strings. The function identifier is *func*_3, and the keyword *func* must precede it. The above function has two child XML elements as its content. The first is the XML element represented by the variable E of type *xml_elem*. The second child XML element is represented by the *<S> F </S>* XML element.

- $xml\_msg\ xml4 : < se > func\_1(se,\ cd1), < se > m\_arr < /se >< /se >$
An XML message declaration. In XML messages, we can use field identifiers like *se* as tag names and content. The above message has the tag

name that is represented by the *se* identifier. In our case, se is the identifier pointing to the string textitmystring. The content of message *xml*4 consists of two XML elements. The first XML element is represented by the XML function func_1 $func\_1(se, cd1)$. The identifiers *se* and *cd*1 are not enclosed in quote characters. That is because we want them to be treated as identifiers of declarations. In these examples, *se* represents a string, *cd*1 represents an XML element. The second XML element is represented by $< se > m\_arr < /se >$. The actual value of xml4 is constructed at a later execution point, when we resolve all identifiers present in it. As before, the identifiers se, m_arr represent declarations: *se* a string, and m_arr is the identifier of a fixed_size array (we omit including the declaration of m_arr).

At the current implementation stage, the XML grammar is merged with the formats grammar. We do not use the same abstract syntax for both XML and formats. However, both XML and format syntax use the same abstract syntax data types (Field type, declared in the file my_parser.y). As seen from the examples above, we can include format fields as XML content.

The implementation performs syntactical correctness checks on XML declarations. So every XML element, function and message declaration has to conform to certain rules:

▷ The XML content has to strictly be in one of the three categories:
1 At least one or more strings and/or string parameters, separated by commas.
2 At least one or more XML elements and/or XML element parameters, separated by commas.
3 At least one or more format fields and/or field parameters, separated by commas.
Mixing any of these three categories is considered malformed content, and terminates the program with an error message.

▷ The values used for tags and attributes must be strings, parameters of type string, or identifiers that resolve to strings. Otherwise, an error is generated, and the program terminates with an error message.

The implementation resolves correctly nested XML elements (that might contain other nested XML elements). Value representation for XML declarations is also implemented. Since we are to perform disjointness checks on XML messages, their value representation is necessary.

We present the value representation of two XML declarations, using the implementation's output. For the simple XML message $xml\_msg\ xml3 :< "now" > "cont" < /"now" >$, the output is seen below:

```
XmlValues "xml3" [[StrVal "<now",StrVal ">",StrVal "cont",StrVal "</now>"]]
```

The data type *XmlValues* represents the value of message $xml3$. It consists only of strings and no nested elements, so only the *StrVal* data constructor (used to store strings) appears.

The implementation's output for the more complicated $xml4$ message is seen below:

```
XmlValues "xml4"
[[StrVal "<myname",StrVal ">",
     ChildXmlElemTagFieldVal [StrVal "<myname",StrVal ">",
                                ChildXmlElemTagFieldVal [StrVal "<Tagname",StrVal ">",
                                                            StrVal "cdcont",
                                                         StrVal "</Tagname>"],
                             StrVal "</myname>"],
     ChildXmlElemTagFieldVal [StrVal "<myname",StrVal ">",
                                Bytes (FixedSize 64) (VarValue {f_type = VarType}),
                             StrVal "</myname>"],
  StrVal "</myname>"]]
```

Apart from *StrVal*, the *ChildXmlElemTagFieldVal* data constructor is used to store the values of the XML child elements. Note that the field array m_arr is resolved into its field value, which is *Bytes (FixedSize 64)*.

9.2 Future Work on XML

A more complete XML support from our implementation would be the support of XML schemas. An XML schema can be viewed as a set of constraints, or a grammar, that an XML message must conform to. These can include constraints on the XML content, the number of attributes allowed in it e.c.t. According to the examined schema, an XML message would be rejected as non-compliant to it, or accepted. Due to the lack of time, this idea could not be implemented. We do not perform sanity checks on the string content for now. Such checks would be the detection of illegal characters inside the XML string content. The integration of XML messages into the compare_msg_values is also needed in order to perform comparisons, but has not been done. Examining XML protocols would also give more insight into the features that should be integrated by our grammar and implementation.

Chapter 10

Protocol Comparison Results

Representing the examined TLS and IKE protocol messages with our language was not a problem. The syntax we have so far covers all messages of both protocols. However, the computational part of the experiments turned out to be a really problematic issue. Unfortunately, the computation times to complete each message set are quite high, let alone combining them. A first cause is the fact that we are examining analytically vectors and variable fields. That is, we consider all their length cases, and subsequently construct multiple possible values from the formats that hold them. This approach has the advantage of giving more accurate results for some field comparisons. However, this comes at the cost of a high computational burden. A second cause is the inherent complexity that some messages have. This is very obvious in IKE messages. Unfortunately, on our tested hardware we were unable to process the exact message representations we wrote for IKE and TLS. The computation could not finish, because the our system would run out of resources and terminate the execution. Instead, we discuss the obtained results from altering the message sets.

10.1 TLS Results

The TLS message set was written using [4], [5] Examining the messages of TLS with the normal vector values was a huge computational burden. This is justifiable by the presence of some quite long vectors. Some TLS messages hold vectors of lengths from zero to 65535 bytes long, which create 65536 different value cases. The worst example is the sub_form named *server_dh_params*, which holds three such vectors. When resizing the vector lengths to almost zero, the computations on the TLS set were almost instant.

We should also mention the fact that we abstracted the content of some vectors. We did that by disregarding the vector's content type. For example, in the TLS specification there exists a vector of length from 2 to 65534 bytes, and its content is declared to be of type CipherSuite. Since the CipherSuite values are also described in the specification, we now have a more restricted range of values for this vector. For simple values this would be feasible to calculate, however for more complicated cases this can become quite hard to handle. Thus, we decided to abstract from the vector payloads, and declared them as free variables.

Our implementation found all TLS messages to be disjoint, when their vector lengths were minimal. This will probably be the case when using the specified vector lengths, however at this point we would have to run experiments on a more powerful machine to reach that conclusion safely.

10.2 IKE Results

The IKE message set was written using [6]. The extensive use of format functions reduced drastically the message size of IKE's representation. Each field in IKE contains a tag that identifies the field that follows it. Thus, every field in IKE was seen as a function: depending on the field sequence, the participating field's tag values are different as well.

IKE turned out to be way more cumbersome to compute. It should be noted that compared to TLS, IKE has way less variable length fields. But even when they were zeroed, the computational burden was still quite high, and computation times were very large. The complexity of the examined messages is the main problem in the case of IKE. One example of that complexity is the presence of messages with eight possible field sequences, due to the used [] operators. Another example are messages which contain security associations, noted as *sec_assoc*. This field contains three nested infinite fields, which also pose a lot of computational burden.

We examined parts of the message set of IKE, but we did not run comparisons on the entire set. Unlike TLS, some messages of IKE were found to be non-disjoint by our checker.

10.3 Examining TLS with IKE

Since our implementation finds some messages of IKE to be non-disjoint, we did not try to compare the message sets of TLS with IKE. A composition of TLS with IKE would not be secure, so there was no point in performing this comparison.

Chapter 11

Conclusions and Future Work

In conclusion, a language has been designed that manages to represent the message sets of TLS and IKE correctly and accurately. Support for XML messages, although not complete, is in an advanced implementation stage for now. Although we abstract slightly from the details of some of the examined messages, this does not reduce the correctness of the representation. Our implementation supports all input that can be generated from our language, with some restrictions posed on the number and placement of encrypted fields. Finally, we managed to perform disjointness checks and get some results on the complex message sets of TLS and IKE. We would be more that interested to try out our implementation on a more powerful machine, in order to reach a verdict on the improvements and changes needed to it.

To improve our current work, first we should incorporate all the XML features we lack for now. Examining more protocols would also result in the expansion of the language syntax, with the aim to create a protocol message description language able to cover as many protocols as possible.

An important consideration is towards which direction the implementation should head. At this point, our implementation is a mix of analytical computation and static analysis. The analytical behavior comes from the treatment of vectors and variable length fields. The treatment of encrypted fields though is different.

Since we abstract from field lengths, we act similarly to the behavior of the $x(Rest)$ field as mentioned in section 3.1.

As seen from the results, the analytical approach tends to be very cumbersome for complex messages. On the other hand, its results are accurate. Computational intensity is problematic, however, not necessarily restrictive. If we are interested to investigate the disjointness of a protocol set, we only need to run the check once on powerful hardware.

A more abstract approach on field comparisons would probably lessen the computational intensity, even though it would still cause heavy branching on the execution trace. When abstracting from details, we overapproximate. And the rationale behind that is to have a non-ambiguous correct result that occurs from examining a more general case.

We believe that this work can aid research done on protocol security, and expand even more in the future. When the unfinished work is completed, the final tool will be useful for both researchers and protocol designers.

Bibliography

[1] Hopcroft, Ullman, Motwani: Introduction to Automata Theory, Languages and Computation 2nd Edition

[2] `http://www.w3schools.com/xml/xml_examples.asp`

[3] `http://www.w3schools.com/xml/xml_syntax.asp`

[4] `http://tools.ietf.org/html/rfc5246`

[5] `http://tools.ietf.org/html/rfc6066`

[6] `http://tools.ietf.org/html/rfc5996`

[7] Baader, Snyder: Handbook of Automated Reasoning, 2001

[8] Arapinis, Duflot: Bounding Messages for Free in Security Protocols

[9] Thomas Groß, Sebastian Mödersheim: Vertical protocol composition (Extended Version)

[10] Aho, Lam, Sethi, Ullman: Compilers, Principles, Techniques, & Tools 2nd Edition

[11] `http://www.haskell.org/haskellwiki/Haskell`

[12] `http://www.haskell.org/alex/`

[13] `http://www.haskell.org/happy/`