University of Edinburgh
Division of Informatics

Elyjah: A security analyzer for Java implementations of communications protocols

4th Year Project Report
Computer Science

Nicholas O'Shea
s0237477

28 February 2006

Communication protocols are consistently studied in computer science due to the ever increasing use of communication. Years after a protocol is published it is not uncommon for a flaw to be found. Static analysis is a tool for finding these flaws. However it is underused by the community at large as most people are unfamiliar with languages that are amenable to this kind of analysis. One possible language for this purpose is the LySa process calculus. Most developers, however, will have experience with an object orientated programming language such as Java. Being able to implement a protocol in Java and having it automatically converted into the LySa language opens up the technique of static analysis of communications protocols to more of the developers who could benefit from it. This report describes a software tool which performs this conversion taking Java implementations and producing a model of the protocol in the LySa language which can act as input to the LySatool. The LySatool analyses protocols relative to an attacker, who attempts to subvert good communication by replaying messages, using any information passed in earlier communication to decrypt or alter later messages or inject messages which he originates.

Acknowledgements

I would like to thank: Stephen Gilmore, my project supervisor, for advice and guidance; John Longley and the other members of my project group, for their continued interest and constructive criticism throughout the intermediate group project meetings; and Mikael Buchholtz et al for their work on LySa and the LySatool without which this project wouldn’t be possible

Contents

1. Introduction

1.1 Project Outline

Communication protocols are consistently studied in computer science due to the ever increasing use of communication. Years after a protocol is published it is not uncommon for a flaw to be found. Static analysis is a tool for finding these flaws. However it is underused by the community at large as most people are unfamiliar with languages that are amenable to this kind of analysis. One possible language for this purpose is the LySa [2, 4, 6, 7, 11] process calculus. Most developers, however, will have experience with an object orientated programming language such as Java. Being able to implement a protocol in Java and having it automatically converted into the LySa language opens up the technique of static analysis of communications protocols to more of the developers who could benefit from it. This report describes a software tool which performs this conversion, taking Java implementations and producing a model of the protocol in the LySa language which can act as input to the LySatool [1]. The LySatool analyses protocols relative to an attacker who attempts to subvert good communication by replaying messages, using any information passed in earlier communication to decrypt or alter later messages or inject messages which he originates.

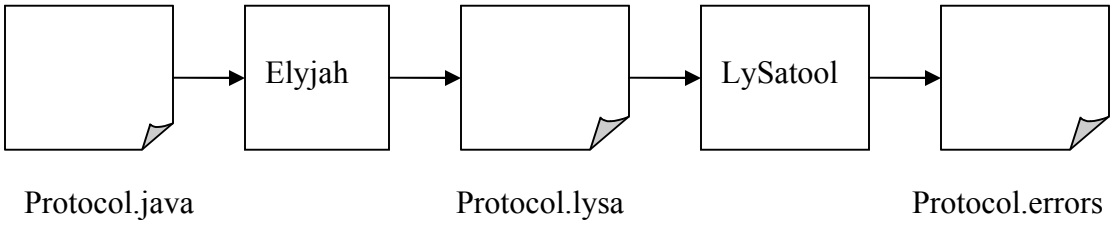

Figure 1 – Computation performed on Java implementation of protocol

1.2 Background Information

This section gives background information that the reader may find useful to understand the following description of the project. The Wide Mouthed Frog protocol will be consistently used as an example in this section to show various ways of modelling protocols.

1.2.1 Security Protocols

A security protocol is, in this context, a series of communications between two or more principals. A principal will usually be an independent communicating entity, linked to other principals through some network. These communications can take place for many reasons, such as authentication, sharing of data (secret or otherwise), a combination of these or some larger task.

There is a lot of interest in security protocols because, despite clever planning, flaws can still be found in them, even though it may take years for this flaw to become known. A flaw in a protocol means that an attacker can either glean secure information or mislead a principal in some way that was not intended.

There is a typical method of easily describing protocols, called protocol narrations. A typical line of one of these narrations could be:

$$A \rightarrow B : A, P$$

This line simply says, principal A sends to principal B a message containing two parts, A's name and a password P. This could be part of a protocol to allow A to get access to some resource on B.

If part of the message is to be encrypted, this can be represented as follows:

$$B \rightarrow A : B, \{B, Msg\}_{KAB}$$

This line states that B sends a message to A consisting of its own name in plaintext and then an encrypted block which can be decrypted using the key Kab. When decrypted the message consists of two parts, B's own name again and some other message. An example of a full protocol narration is given below for the Wide Mouthed Frog protocol.

$$A \rightarrow S : A, \{B, K_{AB}\}_{KAS}$$
$$S \rightarrow B : \{A, K_{AB}\}_{KSB}$$
$$A \rightarrow B : \{mess\}_{KAB}$$

Protocol narrations of this nature have a serious limitation. This method of describing a communication protocol only describes what messages are sent by each principal, omitting instructions to the receiver on what action to take upon receiving each message. A more accurate description of a protocol would include inputs, checks and decryptions. These extra assumptions need to be represented more formally for static analysis of the protocol to be possible. To get an idea of the extra information needed to model a protocol, below is a representation of the Wide Mouthed Frog protocol taken from [6].

1: $A \rightarrow$: $A, S, A, \{B, K\}:_{KA}$ [assuming K is a new key]
1': $\rightarrow S : x_A, x_S, x'_A, x$ [check $x_S = S$; $x_A = x'_A$]
1'': S : decrypt x as $\{x_B, x_K\}:_{KXA}$

2: $S \rightarrow$: $S, x_B, \{xA, x_K\}:_{KXB}$
2': $\rightarrow B : y_S, y_B, y$ [check $y_B = B$]
2'': B : decrypt y as $\{y_A; y_K\}:_{KB}$

3: $A \rightarrow$: $A,B, \{m_1, \ldots, m_k\}:_K$
3': $\rightarrow B : z_A, z_B, z$ [check zB = B; zA = yA]
3'': B : decrypt z as $\{z_1,\ldots, z_k\}:_{yK}$

This additional notation now makes clear the additional work done by the receiver of the message. Like the LySa language explained later, the first two components of the message are the sender and then the receiver. The protocol above still represents three messages but the processing of each message is broken down into three steps. The first represents the sender's actions; the final two steps show the receiver's actions. The second step, represented by a single apostrophe after the message number, details the first step of work the receiver performs upon receiving a message, namely checking certain parts of the message match expected values as well as binding message parts to variables. The third step represented by two apostrophes details the decryptions the receiver has to do to the message.

1.2.2 The Dolev-Yao Attacker

The security of communication protocols is often analysed with regard to a Dolev-Yao attacker[3]. A Dolev-Yao attacker has several abilities:

- Intercept messages
- Decrypt messages
- Create new messages
- Encrypt messages
- Send messages

The goal of the attacker is to breach confidentiality and/or authentication. As well as injecting messages which an attacker originates, it is also possible for an attacker to replay messages that it intercepts between legitimate principals and use any information passed in earlier communication to decrypt or alter later messages.

1.2.3 Encryption and Decryption

The art of encryption has been around since antiquity when primitive methods were used to restrict access to data. At its most basic, encryption is the process of modifying some information so that it is unreadable without some additional knowledge. This extra knowledge is typically a key which is used in the encryption and decryption process. In security circles, the term cryptovariable can also be used instead of key. This represents the fact that the key changes the operation of the encryption algorithm. Modern encryption can be broken down into two distinct groups based on the type of key used, symmetric and asymmetric encryption.

Symmetric Encryption

Symmetric encryption refers to a method of encrypting data where the same key is used for both the encryption and decryption processes. Generally, symmetric encryption is computationally hundreds of time faster than asymmetric encryption. However, to balance this, symmetric encryption is less secure, mainly due to the need for each principal to have knowledge of the same dual purpose key.

Asymmetric Encryption

Asymmetric encryption uses two different keys, one for encrypting data and one for decrypting data. This means that the key used for encrypting data can be known by everyone while the other is kept secret. This allows anyone to send someone an encrypted message and be confident that only the intended recipient can decrypt the data. Because of this difference in the secrecy of the keys, one is known as a public key, the other a private key.

A common analogy used to clarify the difference between these two kinds of encryption is that of sending padlocked boxes between two people, usually named Alice and Bob. Under symmetric encryption, Alice and Bob both have keys to the same padlock. This padlock is used by Alice to lock a box, into which she has already placed some secret information. This box is then sent to Bob, who uses his key to unlock the padlock and read Alice's message. In asymmetric encryption, when Alice wants to send a message to Bob, she must first request Bob's padlock. She uses this padlock to lock a message inside a box, and send the box to Bob who can then use his key to unlock his own padlock and read the message.

1.2.4 Process Calculus

Also known as process algebras, process calculi are a set of formal approaches to modelling concurrent systems. Once modelled in a formal language, the systems can be analysed and deductions made about process equivalence.

While there is a large scope of variety in various process calculi, they all have a foundation of similar features. A common feature is that communication between two processes takes place using message passing as opposed to shared variables. This typically takes place using channels; however as LySa is a notable exception to this, no time will be spent here describing these channels.

Sequential composition is represented by two processes separated by a period. Temporal order is of importance, thus the process A.B will first complete process A before B is activated.

Concurrent processes are separated with a vertical bar. Parallel composition between two processes, P and Q, represented by P | Q, means that both processes can conduct computation as well as communicate with each other.

Replication of processes is signified by an exclamation point.

Termination of a process is represented by a symbol such as a 0, nil, STOP ETC.

1.2.5 LySa

LySa is similar to the more-well known Spi-calculus and π-calculus, although it differs from these two which preceded it in two ways. The first is the absence of channels in the LySa language. This is replaced by the idea that all communication

takes place in a central medium which all processes can interact with. By not representing channels, LySa actually makes itself more representative of networks where attackers can eavesdrop and insert their own messages. This is fortunate as nearly all networks, for example the internet, are of this form. Calculi which rely on channels for communication actually add a level of security to the protocols which is lacking in the real-world. The second difference between LySa and the previously mentioned calculi is that pattern matching is applied to the inputs both of plain-text and encrypted messages.

LySa Syntax

As in any process calculus, LySa is split into terms and processes with keys, messages, nonces etc represented as closed terms.

$E ::=$	*terms*	
	n	name ($n \in N$)
	x	variable ($x \in V$)
	k^+, k^-	public and private keys
	$\{E_1, \ldots E_k\}_{Eo}$	symmetric encryption ($k \geq 0$)
	$\{\|E_1, \ldots E_k\|\}_{Eo}$	asymmetric encryption ($k \geq 0$)
$P ::=$	*processes*	
	0	nil
	$< E_1, \ldots E_k>.P$	output
	$(E_1, \ldots E_j;x_{j+1}, \ldots , x_k).P$	input (with pattern matching)
	$P_1 \mid P_2$	parallel composition
	$(\nu\ n)\ P$	restriction
	$! P$	replication
	decrypt E as $\{E_1, \ldots E_j;x_{j+1}, \ldots , x_k\}_{Eo}$ *in P*	symmetric decryption (with pattern matching)
	decrypt E as $\{\|E_1, \ldots E_j;x_{j+1}, \ldots , x_k\|\}_{Eo}$ *in P*	asymmetric decryption (with pattern matching)

Sending & Receiving Messages with Pattern Matching

As LySa's communication syntax is different to other process calculi some space will be given to running through a brief example. The format for a LySa message is as follows: the first two parts are the source and destination respectively. The remaining parts of the message are either encrypted or plain-text. Additionally, to declare that a new constant is being generated for the process, the restriction process is used. In order to send a simple message consisting of two phrases "nonce" and "msg" from principal A to principal B, the LySa process will read:

(ν nonce) (ν msg) <A, B, nonce, msg>

When a message is received it performs pattern matching on the first portion of the message and binds variables to the other parts of the message in the second

portion. These two portions are distinguished by a semi-colon. It is likely that participant B will wish to perform pattern matching on the first two elements, and possibly on the third if the “nonce” has already been shared between the two. If this is the case the LySa process will read:

(A, B, nonce; msg)

The entire process representing both principals is thus:

(ν nonce) (ν msg) <A, B, nonce, msg> | (A, B, nonce; msg)

Modelling Encryption/Decryption in LySa

LySa models perfect encryption, that is to say it does not consider brute-force attacks possible, and the only way to decrypt an encrypted message is to know the correct key. LySa supports modelling of both symmetric (shared key) and asymmetric (public-key encryption), and thus Elyjah should do also.

Shared Key Encryption

Encryption is added to a send message process by enclosing the encrypted section in curly braces and adding a colon and the key afterwards. For A to send an encrypted nonce and message to B under the key “K”, the process is:

<A, B, {nonce, mess}: K>

Decryption is a two stage process. When the message is received, it will be bound to a variable and in the next process be decrypted. Pattern matching is also employed here, allowing checks to be made on encrypted messages. B’s processes to receive and decrypt the above message would therefore be:

(A, B; x). decrypt x as {nonce; msg}: K in 0.

Public Key Encryption

With public key encryption, keys are declared for encryption or decryption with either a plus or minus sign. Additionally as well as the curly braces, vertical bars frame encryption and decryption. In order to use the restriction process to create a key pair for use with public key encryption the process must read:

(ν +- K)

Crypto-Points

Crypto-points are a vital part of LySa’s ability to analyse protocols. A crypto-point represents a point in the process where either an encryption or decryption takes place. Coupled with a crypto point is an assertion for the origin or destination of the

encrypted message. When part of a message is encrypted, the developer can choose to specify the current location as a crypto-point. This is done by adding the following after an encryption:

[at a]

In the example a can be replaced with any label. Additionally the developer can then choose to specify one or more crypto points where decryption takes place in a valid run of the protocol. This is done as such:

[at a dest {b}]

When a decryption takes place, a crypto point is marked as above. The crypto point where the message was encrypted in a valid run of the protocol is represented like the example above but with 'orig' taking the place of 'dest'.

1.2.6 Static Analysis

Once a protocol is represented in LySa, it can be analysed through static analysis of the LySa code. The term *static analysis* means that the analysis is performed without running the code in any way. Static analysis is increasingly recognized as a fundamental tool for program verification, bug detection, compiler optimization, program understanding, and software maintenance. The LySatool is one example of static analysis; another is Elyjah itself, which will take the Java source code and convert it into the LySa syntax.

1.2.7 LySatool

The LySatool is an automatic tool for checking the security properties of protocols. The tool takes protocols modelled in the LySa process calculus. It then provides feedback regarding which parts of the protocol are transmitted in secret as well as whether an attacker can falsely achieve authentication by inserting messages at any point. An important point to mention is that the analysis is an over approximation of the actual behaviour thus can guarantee confidentiality and authentication properties. The downside of this approximation is that the tool will occasionally report faults due to attacks that are impossible to reproduce in a real world scenario.

When reporting violations of authentication properties, the LySatool uses the term CPDY to signify a crypto-point in the Dolev–Yao attacker. To represent that an attacker can decrypt an encrypted message the LySatool will list under the violation of authentication properties:

(a, CPDY)

Here 'a' is a valid crypto-point, specified by the developer of the model. Additionally an attacker may be able to encrypt some message which will then be decrypted by a principal as part of a protocol. This will show up as a violation represented by:

(CPDY, b)

Full details of how the LySatool works can be found in [6].

1.2.8 For- LySa

For-LySa [5, 10] is a tool that addresses the same problem that this project aims to help rectify, that of allowing non-experts to use the LySatool to analyse security protocols. This tool allows a user to design a model of their protocol in UML which is then converted into LySa. Much like the scope of this project the input must conform to a fixed framework. However, while For-LySa takes a UML representation of a protocol, Elyjah will use a Java program as its input.

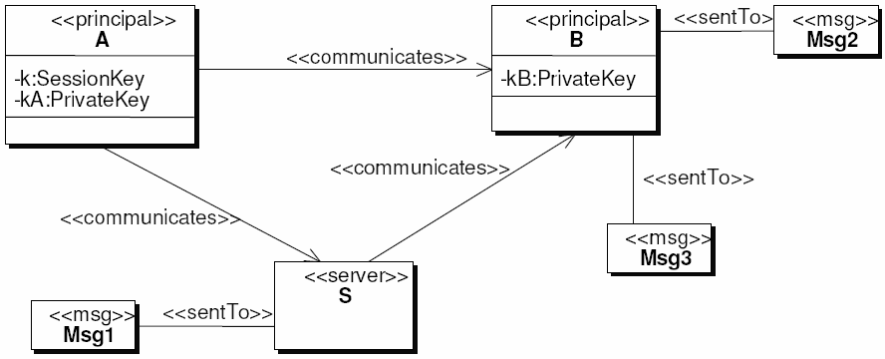

Figure 2 Overview of the Wide Mouthed Frog Protocol as modelled in For-LySa. From [10]

The above shows a UML representation of the Wide Mouthed Frog protocol and below is the structure of one of the messages.

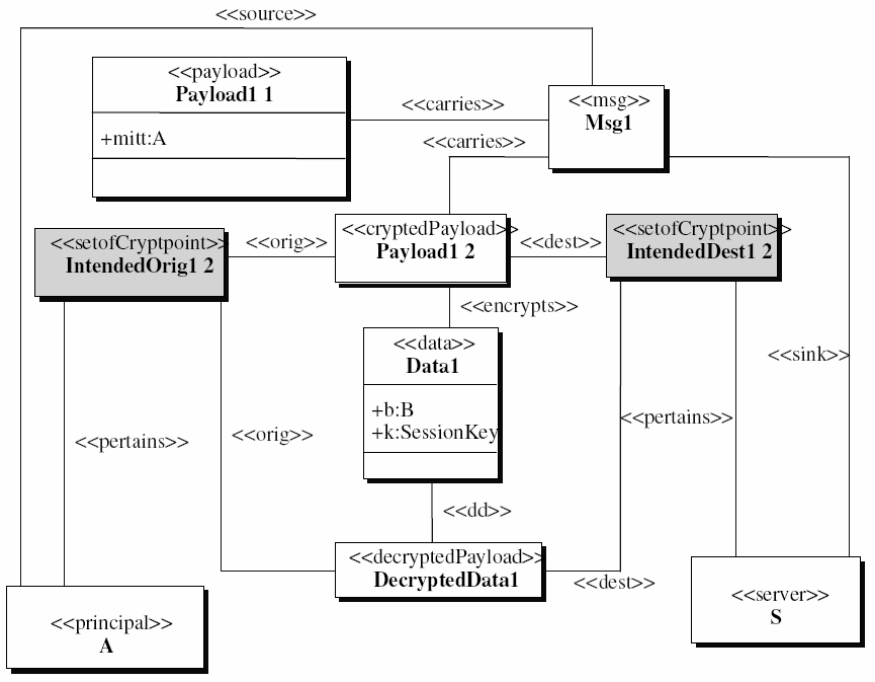

Figure 3 Structure of Msg1 from WMF protocol. From [10]

It is the belief of the author that while this may be easier to understand than a plain LySa representation, the number of developers unfamiliar with UML is sufficient enough for it to be worth exploring other input formats.

Another advantage of the proposed software tool is that it will allow a developer to actually run the protocol and see output at various stages. This guarantees that the input is not a misrepresentation of the protocol. Another major disadvantage of For-LySa is that it only works with shared key encryption severely limiting the protocols that can be modelled with it.

1.2.9 The Java Programming Language

In order to model a protocol in Java, it is required to examine Java's security credentials. The most important thing to examine is the cryptography possibilities. Java supports both symmetric and asymmetric encryption as well as digital signatures, message digests and message authentication codes. It is worth remembering that Java is a lot more expressive than LySa, as such only a portion of possible Java inputs are needed to represent the entirety of LySa's expressive capabilities.

Java has a class called Key which provides an interface for multiple sub-interfaces and subclasses. In order to represent the various methods of encryption that LySa provides multiple types of Key are required. In order to represent symmetric encryption, an implementation of the SecretKey interface is required but asymmetric encryption will require a public/private key, generated from some key pair. The algorithm used is of course irrelevant as LySa has no representation of algorithmic behaviour or data. In fact, as LySa models any encryption as perfect encryption there is no need for any real encryption in the model. Although for debugging reasons as well as purity it is worth really encrypting messages as required. The class diagram for the Key objects that will be used in this project is given below.

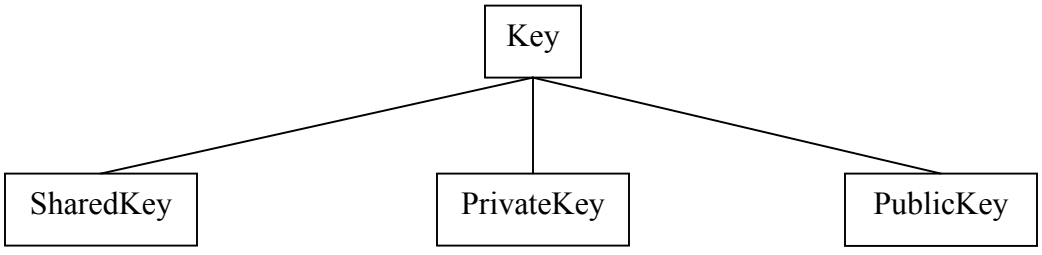

Figure 4- Class diagram of Java's Key interface and subinterfaces

Implementing each of these three interface nodes are various classes which use different algorithms to implement these interfaces. In order to generate an instance of a SecretKey using the DES algorithm, a key generator first has to be initialised, then used to generate a key as below.

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES");
SecretKey key = keyGen.generateKey();
```

In order to generate a key pair, for use in asymmetric encryption a key pair generator must first be created and initialised. The following code block gives details on creating 1024 bit DSA public and private keys.

```
KeyPairGenerator keyGen =
     KeyPairGenerator.getInstance("DSA");
keyGen.initialize(1024);
KeyPair keypair = keyGen.genKeyPair();
PrivateKey privateKey = keypair.getPrivate();
PublicKey publicKey = keypair.getPublic();
```

To encrypt a String object a cipher has to be created and initialised with a key. The encryption itself works on a byte array so the String must be converted to an array of bytes and then back again the encryption has been performed.

```
ecipher = Cipher.getInstance("DES");
ecipher.init(Cipher.ENCRYPT_MODE, key);
byte[] utf8 = str.getBytes("UTF8");
byte[] enc = ecipher.doFinal(utf8);
String encrypted =
          new sun.misc.BASE64Encoder().encode(enc);
```

Finally to decrypt a string, the same key must be used to generate a decryption cipher. As with encryption, the string must be converted to a byte array, and then back into a string after decryption has been performed.

```
dcipher = Cipher.getInstance("DES");
ecipher.init(Cipher.ENCRYPT_MODE, key);
dcipher.init(Cipher.DECRYPT_MODE, key);

byte[] dec =
   new sun.misc.BASE64Decoder().decodeBuffer(str);
byte[] utf8 = dcipher.doFinal(dec);
String message = new String(utf8, "UTF8");
```

2. Java Input Format

Although it would be possible for a software tool, such as Elyjah, to be able to accept any Java program as input before converting the source code into a LySa process, it would be beyond the scope of this project to achieve this. Such a tool would also require much more computational power and is likely to be substantially less reliable than one which demands a uniform input format. Thus a framework is needed to allow developers to model protocols, while also allowing the software tool to parse the source code and identify the security-critical operations. An implementation of a protocol needs to be a fully-working Java program which the developer can use to test that the protocol functions as expected before analysing its security properties. In order for it to be possible for any tool to understand the developer's intent, there needs to be a standard method of achieving certain goals. As such there needs to be set ways of encrypting/decrypting data as well as sending messages between principals.

2.1. Design

Each principal must be modelled in a separate class which extends the Thread class thus allowing multiple principals to be running at the same time, thereby simulating LySa's concurrent processes. These classes must all be within the same Java file, thereby helping Elyjah as it will only have to parse a single file. There will also need to be a class in this same file whose `main` method contains the code to set up the principals; create any keys known prior to the commencement of a protocol; and finally establish the network through which all principals will communicate. This network will also need to be implemented as a separate class which is capable of keeping references to the various principals. Finally, there needs to be a class for generating keys as both the principals and protocol initialisation class will need to be able to do this and should use the same implementation to do so.

There needs to be an abstract class set up which implements most of the functions of a principal while leaving un-implemented the `run` method (inherited from the Thread class) and a method to deal with an incoming message. The `run` method will be used to start the protocol, for example, if principal A sends a message in a protocol, then A's `run` method will contain this code. Other functions of the class that are required would be: to organise a key store for the three possible kinds of key: SecretKey, PublicKey and PrivateKey; and to encrypt and decrypt blocks of messages, allowing for crypto point annotation at these points. These methods will be used as keywords so that the parser can pick out the key parts of the protocol. As a developer is likely to want to be able to print out the values of variables and other debugging information, a Logger should be set up in the abstract class to encourage developers to use this rather than the less informative `println` system call.

In addition, a uniform method of dealing with incoming messages is needed. The proposal for this project is to use case/switch blocks. A variable stores which message is expected next and each time the processIncoming method is called this

variable is incremented. Pattern matching is modelled by using assertions to check that the value at each position of the incoming vector matches expected the one.

The network class only needs methods to send a message from one principal to another and some method of registering principals so that the correct class' incoming message method can be called when required.

The key generation class needs to have a method to return a SecretKey for symmetric encryption and another method to return a KeyPair for asymmetric encryption.

Messages will be a vector of strings. The send command will need to have an argument for this vector, the receiver and the current class (in order that the receiver can work out who the sender is).

Encrypt and decryption methods will also work on vectors of strings and will require optional arguments for crypto-points. This will require several different overloadings of these methods as it is possible for the crypto point to specify only the current location, one origin/destination, multiple origins/destinations or neither.

Method headers for all the required classes can be found in the appendix.

2.2. Implementation

This section will give details of the implementation of the Java framework that will be used to model protocols so that Elyjah can convert them into LySa processes.

2.2.1. Network Class

The network class is primarily used to pass messages between principals. It is also used to pass any keys that need to be known by multiple principals before a protocol begins. In order to do this there needs to be a store of the classes and a way to convert between the classes and the names used to identify them. This conversion needs to take place in both directions. This is achieved through two Hashtables set up using generics

```
private Hashtable<String, ComClass> classNameToClass;
private Hashtable<ComClass, String> classToClassName;
```

When the class is created, the constructor will initialise these Hashtables. When a call to the `register` method is made, supplying a class and a name supplied as a string, entries are made in both of these Hashtables.

When a principal needs to send a message to another principal, the `send` method is used.

```
public void send(ComClass source, String dest, Vector tuple){
```

```
        ComClass destClass = classNameToClass.get(dest);
        tuple.add(0, classToClassName.get(source));
        tuple.add(1, dest);
        destClass.processIncoming(tuple);
}
```

As well as calling the `processIncoming` method of the correct class, this method also appends two extra message parts to the beginning of the tuple of messages. The first is the sender of the message and the second is the receiver. This puts the message in line with a LySa process.

The class also has a `shareKey` method used to call the `shareKey` method of each registered class. This is achieved through setting up a ListIterator of the key set of classToClassName Hashtable.

2.2.2. ComClass Class

This is an abstract class that the principal classes extend. It provides much of the implementation of the class while also requiring that the class implements the `processIncoming` method.

```
public abstract void processIncoming(Vector<String> v);
```

The use of generics here means that when examining elements of the vector, no cast operations are needed. However it does require that all elements are String objects. This is a problem if the protocol requires sending a key between principals. For this reason the class contains methods to convert a Key object into a String object, and then methods to convert a String object into either a SharedKey or a PublicKey object. This is not a limitation enforced solely by this model however. In real protocols, much like this model, keys need to be converted into a transmissible format. In order to turn a Key into a String, the object must be turned into an array of bytes which can then be turned into a String.

```
byte[] keyBytes = key.getEncoded();
String keyStr = new String(keyBytes, "ISO-8859-1");
```

To convert a String into the correct type of Key requires different methods. To convert a SharedKey back from a String requires converting the String into a byte array and then creating a key specification from this byte array.

```
byte[] keyBytes = str.getBytes("ISO-8859-1");
SecretKey key = new SecretKeySpec(keyBytes, "DES");
```

To generate a PublicKey requires creating a key specification from the string representation of the key, creating a KeyFactory object with the correct algorithm and then generating a public key from this KeyFactory using the key specification.

```
X509EncodedKeySpec pubKeySpec =
```

```
        new X509EncodedKeySpec(keyStr.getBytes("ISO-8859-1"));
    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    PublicKey key = keyFactory.generatePublic(pubKeySpec);
```

The class also has a method to receive a Key directly called `registerKey`, which is to be used when a Key has already been shared before a protocol begins. The method should only be called from within the protocol initialisation class as the method does not map to any LySa process. This method adds a key and an accompanying key name into a key store which is implemented as a Hashtable which uses generics to restrict the Hashtable object's key to a String and the returned object to a Key, This means that when retrieving a key from the key store a cast will still be needed to cast down to the correct type of key. Unfortunately this is unavoidable as the key store is only homogenous to the parent class of Key.

```
private Hashtable<String, Key> keys = new Hashtable<String, Key>();
```

The class also contains an instance of a KeyGenerationClass class as well as methods which use this to create SecretKeys and KeyPairs. These methods call the appropriate method in the KeyGenerationClass.

One of the biggest tasks that this class needs to implement is encryption and decryption. As this has to be possible in both symmetric and asymmetric modes this forces the methods to accept a Key object. The first task in encryption is to turn the Vector object into a String object. This is achieved by using an Iterator to concatenate the parts of the vector together with a separator between them, such as a new-line character.

```
String message = null;
ListIterator i = v.listIterator();
while (i.hasNext()) {
     String msgPart = (String) i.next();
            if (message == null){
                message = msgPart;
            } else {
                message = message.concat(msgPart);
            }
            if (i.hasNext()){
                message = message.concat("\n");
            }
}
```

Once as a single String, the message can be encrypted as described in section 1.2.9. Decryption will then convert the encrypted string into a decrypted string using the method also described in this section. From this a Vector object can be constructed by using a StringTokenizer object to separate the string by uses of the new line character, which was inserted when originally converting the Vector into a String.

```
StringTokenizer st = new StringTokenizer(parts, "\n");
```

```
Vector v = new Vector();
while (st.hasMoreTokens())
     v.add(st.nextToken());
return v;
```

Methods are also provided which include parameters for the crypto-points, however these are not used in the Java code and these methods call the normal encrypt/decrypt methods as required.

The class also provides a method, `check`, for checking that two strings are equal, used for the pattern matching in conjunction with `assert` statements. As well as returning whether two String objects are equal or not, the method also uses the Logger to report on cases where the two String objects are not equal. The Logger object is set up in this abstract class so that the principal classes which extend it have easy access to it.

2.2.3. KeyGenerationClass Class

This class contains only two methods. One for generating a SecretKey object, called `generateSharedKey`, and another for creating a KeyPair object from which a PublicKey and an accompanying PrivateKey object can be created, called `generateKeyPair`. A SecretKey generation is quite simple ,requiring only one line of code:

```
SecretKey key = KeyGenerator.getInstance("DES").generateKey();
```

To generate a KeyPair requires a bit more computation, requiring KeyGenerators and SecureRandoms to be set up first.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
SecureRandom random =
              SecureRandom.getInstance("SHA1PRNG", "SUN");
random.setSeed(userseed);
keyGen.initialize(1024, random);
KeyPair keypair = keyGen.genKeyPair();
```

2.2.4. Example Protocol File

Although the abstract class forces the developer to implement a `processIncoming` method it cannot force the internal workings. Rather a framework has to be self-enforced by the developer. The first requirement is that switch/case statements have to be used to separate processing of different incoming messages. The other requirement is to use the available methods where possible and to use string literals in the message construction wherever the values are important. If a string literal is not used, the name of the variable will be used in the LySa process instead.

An example of a simple protocol can be found in the appendix as MultipleSendReceive.java. This will be used to examine a typical protocol file. The first part to examine is the set up class:

```
public class MultipleSendReceive {
    public static void main(String[] args) {
        Network net = new Network();
        A a = new A(net);
        net.register("A", a);
        B b = new B(net);
        net.register("B", b);
        a.start();
        b.start();
    }
}
```

This method initialises, registers and starts two principals. Principal A contains a run method typical of a simple protocol.

```
public void  run(){
     Vector v = new Vector();
     v.add("password");
     String msg = generateMessage(“Message1”);
     v.add(msg);
     e.send(this, "B", v);
}
```

This sends a simple message containing a plaintext password and a message. This is sent to principal B. A will later send another message to B, with a similar format. This means B has to be able to accept multiple messages, for this reason it is worth looking at principal’s B `processIncoming` method.

```
public void processIncoming(Vector<String> v){
    switch (receivedNum) {
        case 0:
            receivedNum++;
            // check first
            assert check(v.elementAt(0), "A");
            // check second
            assert check(v.elementAt(1), "B");
            // check third
            assert check(v.elementAt(2), "password");
            // assign fourth
            String msg = v.elementAt(3);
            theLogger.info(msg);
            Vector v2 = new Vector();
            v2.add("password2");
            String msg2 = generateMessage("Message2");
            v2.add(msg2);
            net.send(this, "A", v2);
```

```
                break;
            case 1:
                assert check(v.elementAt(0), "A");
                assert check(v.elementAt(1), "B");
                assert check(v.elementAt(2), "password3");
                String msg3 = v.elementAt(3);
                theLogger.info(msg3);
            break;
        }
    }
```

Encryption is easy to add on to this framework. If there is already a key shared between two principals before the protocol begins, the following lines of code must be added in the `main` method of the MultipleSendReceive class.

```
        SecretKey key = generateSharedKey();
        net.shareKey(key, "K");
```

Here is an example of a run method with encryption added.

```
    public void  run(){
        Vector v = new Vector();
        v.add("A");
        v.add("B");
        Vector vToBeEncoded = new Vector();
        SecretKey keyAS = (SecretKey) keys.get("KAS");
        String msg = generateMessage("Encrypted Message");
        vEncoded.add(msg);
        v.add(encrypt(vEncoded, keyAS, "a1", "s1"));
        net.send(this, "S", v);
    }
```

The accompanying receive process for this will be as follows.

```
public void processIncoming(Vector<String> v){
    switch (receivedNum) {
        case 0:
            receivedNum++;
            assert check(v.elementAt(0), "A");
            assert check(v.elementAt(1), "S");
            assert check(v.elementAt(2), "A");
            assert check(v.elementAt(3), "B");

            SecretKey key = (SecretKey) keys.get("KAS");
            String msgToBeDecoded = v.elementAt(4);
            Vector<String> decode =
                decrypt(msgToBeDecoded, key, "s1", "a1");

            String message = decode.elementAt(0);
```

```
            }
        }
}
```

Using this framework, all manner of protocols can be modelled, while still doing so in a manner that will allow a systematic analysis of the source code to extract a formal representation of the protocol.

3. Elyjah Design

The first step in the conversion from the Java source code, as defined above, to a valid LySa process is to create a parser for the Java language which will return an abstract syntax tree for a Java source file. Following this the source will need to be analysed to create a mapping of variable names to types within a context. This will be required as a syntactical representation of a source code will not contain typing information for the variables that were initialised earlier in the program. It will then be necessary to be able to search this store, not only to find out what the type of a certain variable is, within a given context, as well as being able to find any variables of a certain type, within a given context.

3.1 Removal of extraneous activity

The first task that needs to be done after generating an abstract syntax tree in order to ease the conversion process is to remove any code that is not directly influential to the communication protocol. The first and most obvious extraneous code that needs to be removed is any debugging information. As the superclass for a principal sets up a logger, the first thing that needs to be removed is any reference to this logger. Following this, alternative options for debugging should also be removed such as `println` method calls.

The method of removal depends on the implementation but it is expected that the abstract syntax tree will be easier to amend than the original source code itself. With this in mind it would be expected that to remove a block statement would amount to setting a variable value to null.

3.2 Parsing of main class

3.2.1 Identify class names

The names of the class given in the source are not necessarily the same as the names they will be referred to in the protocol. For this reason it is required to keep a record of what the classes are registered as in the network class. This will require searching through the main class to find all calls to the network's `register` method and creating a mapping between the name provided and the name of the class itself.

3.2.2 Identify keys shared before protocol begins

This process will involve searching through the syntax tree of the main class for all calls to methods involving key generation, this will mean either `generateSecretKey` or `generateKeyPair`. From this the LySa processes can be written. In both message and key generation the LySa term is the Greek alphabet symbol for Nu, ν. Thus either ν or the easier to understand English equivalent new can be used, however it is worth noting that the LySatool will only accept *new* as a

valid keyword. From now on in this report, new will be used by convention. This conversion will thus need to take Java and output LySa like below.

`SecretKey key1 = ***generateSharedKey***();` `net.***shareKey***(key1, "KAS");`	(new KAS) or (ν KAS)

3.3 Parsing of principal classes

3.3.1 run method

Each other class is then parsed in turn. The first part of a class that needs to be checked will be whether the class contains a `run` method. It is expected than in any given protocol only one class will contain this method although all classes will be checked. If the method does exist, then the code will need to be searched for any key generation or message generation methods. Key generation methods are as above with the added use of asymmetric encryption. In this instance the LySa process representing this reflects the use of asymmetric encryption by showing the key as:

(new +- KAS)

Additionally to show the use of a message or nonce the `generateMessage` method should be invoked. After checking the code block for any method calls to either `generateMessage` or a key generation method, the code block should be searched for a call to the network class' `send` method. This line can then be parsed to identify the intended receiver and the name of the vector containing the message itself. The class name is used to identify the sender by mapping it to the name registered with the network as discussed in section 3.2.1. The code block is then searched for all references to the `add` method of the Vector to work out what the message actually consists of. Some of the strings added to this Vector may be a representation of an encrypted set of strings. In this scenario, further computation is needed as discussed in section 3.3.2. This process needs to be done for every `send` command in the method as some protocols may rely on multiple messages being sent off before a reply is expected.

Thus an example conversion from Java to Elyjah for a send command would be:

`// Inside A`
 	
`Vector v = ***new Vector***();`
`v.***add***("A");`
`v.***add***("B");`
`net.***send***(***this***, "S", v);`
	<A, S, A, B>

Additional steps must be taken in the Java when sending certain message parts. As mentioned in section 2 a key must be first turned into a String, thus the method call to add a key to the message would be:

```
v.add(sendKey(keyAB));
```

However this is only a variable name of the key, not the name used to identify it within the protocol. Thus when this method call is used in a code block, a search must be made to find the name of the principal by searching for a call to the `get` method of the key store keys.

`// Inside A`
 	
`SecretKey keyAB =`
`  (SecretKey)keys.get("KAB");`
 	
`Vector v = new Vector();`
`v.add("A");`
`v.add("B");`
`v.add(sendKey(keyAB));`
`net.send(this, "S", v);`
	<A, S, A, B, KAB>

3.3.2 Encryption

There are several parts of the conversion where encryption may be used, namely when parsing both the `run` and `processIncoming` methods of each class. For this reason the construction of the LySa process for encryption should be provided as a separate method. Recall that as well as the parts of the message that are to be encrypted enclosed in curly braces, the LySa process for encryption also requires the key and optional crypto points to be included. When encryption is being used, a call to the `encrypt` method will be present instead of a String variable or String literal inside the Vector object's `add` method. This method call will give the key and crypto points and the Vector containing the plain text message. For example:

```
v.add(encrypt(vToBeEncoded, key, "a", "b"));
```

From this line of code the crypto points will be trivial to extract, being the last two parameters of the `encrypt` message call. In order to work out the contents of the encrypted block, a similar strategy to the method of constructing the LySa `send` method is used. Thus the current block of code must be searched to find all calls to the `add` method of the Vector which is to be encrypted, in this case the Vector vToBeEncoded. However further searching is required to identify what the name of the key is as opposed to the name of the variable which represents it. This will mean searching the code for a call to the key store's `get` method which assigns the key to this variable. This value can then be used to add the key used in the protocol. The total code and accompanying LySa process are as follows:

`// Inside A`
`Vector v = new Vector();`
`Vector vToBeEncoded =`
`  new Vector();`
`String msg1 =`
`  generateMessage("Message 1");`
`String msg2 =`
`  generateMessage("Message 2");`
`vToBeEncoded.add(msg1);`
`vToBeEncoded.add(msg2);`
`SecretKey key =`
`  (SecretKey)keys.get("KAB");`
`v.add(encrypt(vToBeEncoded, key,`
`  "a", "b"));`
`net.send(this, "B", vB);`
	(new msg1) (newmsg2)
	<A,B,{msg1,msg2} : K [at a dest {b}] >

In cases where asymmetric encryption is being used, the key will end with a plus symbol. In these cases, the curly braces should additionally have vertical bars inside of them such as the example below.

<A,B,{|msg1,msg2|} : K+ [at a dest {b}] >

3.3.3 processIncoming method

This method will be split into switch blocks so it will be easiest to parse each switch block in turn. Normally each switch block will contain the code for processing one incoming message as well as sending one or more messages to another principal.

The first task is to check for key and message generation, only then can the computation proceed to composing the receive LySa process. This will be a mix of identifying assert statements, variable declarations and decryption. The `assert` statements are used to model LySa's pattern matching, a typical `assert` line will look like this:

```
assert check(v.elementAt(0), "S");
```

The only part of this line which will be used in the LySa process is the second argument of the check method. After all of the assert statements are dealt with, the LySa process is split with a semi-colon before the variable binding portion. In the Java model this is represented by a line such as:

```
String msgToBeDecoded = v.elementAt(2);
```

From this line, the only portion to appear in the LySa model is the name of the variable, in this case the String called msgToBeDecoded. A typical Java model representation of dealing with an incoming message compared to the corresponding LySa process will thus resemble:

`assert check(v.elementAt(0), "S");`
`assert check(v.elementAt(1), "B");`
`String msgToBeDecoded =`
`v.elementAt(2);`
	(S, B; msgToBeDecoded)

As hinted by the name of the variable used above, typically decryption will have to be performed after receiving a message. Just as with receiving a message, pattern matching is employed and implemented in the Java model in the exact same way. The first step is still to decrypt the message, turning it from a string into a vector. This is achieved in a command such as:

```
Vector<String> decode =
          decrypt(msgToBeDecoded, key, "b1", "s2");
```

This relies on the variable msgToBeDecoded being set up as above, and a key retrieved from the key store by means of a `get` method call. The string used to retrieve the key is the one used in the LySa process. The same technique used with the pattern matching above is used to construct the breakdown of the message. It is also possible that the origin crypto point will be a vector as opposed to a string literal. In this case, the LySa process must include all the constituent parts.

SecretKey key = (SecretKey) keys.*get*("KBS"); Vector<String> decode = *decrypt*(msgToBeDecoded, key, "b1", "s2"); *assert check*(decode.*elementAt*(0), "A"); String message = decode.*elementAt*(1);	decrypt msgToBeDecoded as {A; message}:KBS [at b1 orig{s2}]

Like encryption, if asymmetric encryption is being used then the curly braces must additionally have vertical bars on the inside. After constructing any decryption processes that need to be done, the switch block may include some send commands, if this is the case the LySa processes for these will need to be constructed in the same manner as discussed in section 3.3.1.

3.3.4 Summary of Conversion

To re-iterate the process contains the following steps

1. Remove all traces of extraneous activity such as logging /debugging information.
2. Identify any keys created in the main class.
3. Process `run` method.
    - Add any key/message generation in the method
    - For each send command work out the appropriate LySa command
4. Process `processIncoming` method.
    - For each switch block: add any key/message generation
    - Work out appropriate LySa command receive message
    - Work out any decryption that is necessary

- For each send command work out appropriate LySa command.

4. Implementation

In order to implement this system within the allotted time frame, a pseudo extreme-programming methodology was used. This meant having fast design, implement, test cycles; fixing bugs when they were discovered; and only implementing the minimal functionality needed at each point.

4.1. The Java parser

Before working on the conversion process itself, it is necessary to create a Java parser. There are a couple of Java parser generators in wide circulation, namely CUP and JavaCC. At first this project attempted to use the CUP parser generator. This generator was initially recommended due to its use in some of the department's taught courses. However while learning about and testing the parser generator with simple grammars worked fine, when attempting to create a parser using the full Java grammar supplied on the CUP website errors were generated.

```
Opening files... Parsing specification from standard
input... Error Syntax error at Symbol: SUPER in line 196 /
column 33 Error Illegal use of reserved word Error Syntax
error at Symbol: SUPER in line 534 / column 1 Error Illegal
use of reserved word Error Syntax error at Symbol: SUPER in
line 537 / column 4 Error Syntax Error Closing files... ---
---- CUP v0.10k TUM Edition 20050516 Parser Generation
Summary ------- 6 errors and 0 warnings 106 terminals, 218
non-terminals, and 535 productions declared, producing 0
unique parse states. 0 terminals declared but not used. 0
non-terminals declared but not used. 0 productions never
reduced. 0 conflicts detected (0 expected). No code
produced. ----------------------------------------------
---
```

After investigation it was revealed that there is an incompatibility between the latest version of the CUP parser generator and the Java grammar file supplied. This is due to the latest version of the CUP parser generator including support for generics which was not available in previous versions, significantly the version the Java grammar file was written for. This causes a problem as the grammar now supports types such as:

ArrayList<? super Container>,

This means that super is now a keyword, resulting in the errors seen above. Thus the workarounds involve either using a previous version of the parser generator or changing all uses of the keyword super within the Java grammar file and all inputs to

avoid the clash. Both of these options were less than optimal, so it was decided to try using the JavaCC parser generator instead.

Just like the CUP parser generator, the JavaCC parser generator's website contains a file for the Java grammar. However in this project a parser is not sufficient, a means of navigating an abstract syntax tree is also required. JavaCC by itself will only return whether an input file is successfully parsed. There is an add-on for JavaCC known as JJTree which allows the generated parsers to produce syntax trees. In order to generate a parser to do this there are several stages of computation. First a JJTree file is converted to a JavaCC file which is then converted to a java source file before finally being converted to a class file by the normal Java compiler.

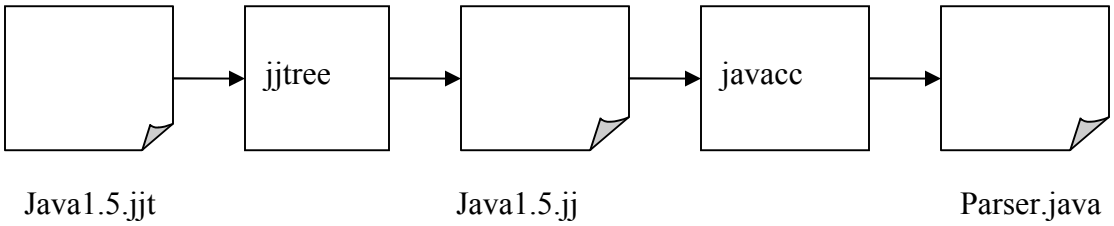

Figure 5 - Parser Generator Construction Process

The supplied Java grammar file from the website comes in the form of a JavaCC file which had to be converted into a file suitable for JJTree by modifying the parser code in the grammar file.

JJTree defines a Java interface, Node, which all parse tree nodes must implement. This interface provides methods for navigating through a tree of nodes using methods to traverse to the parent as well as through the children of a node. These methods are called `jjtGetParent` and `jjtGetChild`, this method takes as an argument an integer used to specify which child to traverse to. This interface is implemented by a SimpleNode object which is created automatically by JJTree. This class can then be extended or modified as needed. For this project, the class was modified to increase the error handling capabilities and change the return values of the tree traversal methods from Node to SimpleNode to eliminate the need for casting in Elyjah.

Using JJTree, after a source file is parsed the root of the abstract syntax tree is returned as a SimpleNode. It is then possible to navigate the abstract syntax tree much as any other tree. Classes, methods and even single lines of code have a tree structure so a single SimpleNode can be used as a pointer to any part of the code.

4.2. Searching the abstract syntax tree

It will be necessary to search through the abstract syntax tree in order to find certain nodes. This will be done by iterating through the children of a node and running the method on each of them in turn. For an example see the imaginary method below.

```
public static void methodName(SimpleNode node){
      if (node.toString().equals(nodeName) {
            // Process Node
      } else {
            for (int i = 0; i < node.jjtGetNumChildren(); ++i) {
                  SimpleNode n = node.jjtGetChild(i);
                  if (n != null) {
                        methodName(n);
                  }
            }
      }
}
```

Many of the methods described below will take the form of the above template. This method is called on the root of some block of code, such as a single line, a method, a switch block or a whole class. The current node is then checked to see if it contains the required string, nodeName in the above example. If so then some additional processing will be performed else this method is then called on the children of the node. The values generated by methods of this sort are usually stored in static Vectors allowing the original method to create Iterator objects to iterate through the values of the Vector.

4.3. Creating a XML representation of the abstract syntax tree

In order to examine the abstract syntax tree as well as store trees for later analysis, a method was developed which would print out the abstract syntax tree as an XML file. This allows easy analysis of the abstract syntax tree and storage of entire programs in the XML representation. This takes the form of the template given above but instead of searching for a node with a particular value, every node will be processed.

```
 public static void dumpXML(SimpleNode node, String pre) {
       if (node.jjtGetNumChildren() == 1) {
            System.out.println(pre+"<"+node.toString()+">");
            dumpXML(node.jjtGetChild(0), pre + " ");
            System.out.println(pre + "</"+node.toString()+">");
       } else if (node.jjtGetNumChildren() != 0) {
            System.out.println(pre+"<"+node.toString()+">");
            for (int i = 0; i < node.jjtGetNumChildren(); ++i) {
                  SimpleNode n = node.jjtGetChild(i);
                  if (n != null) {
                    dumpXML(n, pre+" ");
                  }
            }
            System.out.println(pre+"</"+node.toString()+">");
       }else{
            System.out.println(pre+"<"+node.toString()+" />");
       }
}
```

4.4. Retrieving class names

As each class will later be parsed individually it will later be necessary to possess a list of the classes in the input file along with pointers to the nodes at the top of these blocks' syntax trees.

A class name can be found by searching the syntax tree for a node containing the string ClassOrInterfaceDeclaration. Once this node is found the name of the class is located in the first child of this node, such as below.

```
<ClassOrInterfaceDeclaration>
    <A />
```

As an example of how this searching and tree traversal is implemented, the first few lines of this method are given. If the class name is the same as the name of the file, then it is marked as the protocol set-up class and is parsed differently. The rest of the method follows the template given in Section 4.2.

```
public static void getClasses(SimpleNode node){
    if (node.toString().equals(nodeName) {
      String className = node.jjtGetChild(0).toString();
      classes.put(className, node);
```

As discussed in Section 3.2.1, it is also necessary to have a mapping between the name of classes and the name used to register them with the Network class. This is achieved by searching the main method of the protocol set-up class for all calls to the `register` method of the Network class. Both arguments from this method call should then be extracted and used to create a HashTable using the actual class name as the key and the String used to register it as the object. An additional HashTable is also used to map between the class name and the root of the class.

4.5. Gathering type information

Once the abstract syntax tree has been generated, it is necessary to create a mapping of variable names to types within certain contexts. For this, several classes were created. The first is a class, Variable, which is designed to store the following details of a single variable:

- Variable name;
- The type of the variable
- Name of the method the variable is in, (Global if not in a class)
- Name of the class the variable is in

The second is a class, VariableStore, which contains a vector of variables as well as methods to add a variable to the store and various methods to search through the vector. These methods return the type of a variable given its name and location, the name of any variables of a given type within a certain location as well as all the variables in a method. In order to populate this store, the abstract syntax tree is

parsed to look for any variable declaration block. An XML representation of a LocalVariableDeclaration block statement looks like:

```
<BlockStatement>
        <LocalVariableDeclaration>
                <ReferenceType>
                        <PrivateKey />
                </ReferenceType>
                <VariableDeclaratorId>
                        <privateKey />
                </VariableDeclaratorId>
                <PrimaryPrefix>
                        <Name>
                                <keypair />
                                <getPrivate />
                        </Name>
                </PrimaryPrefix>
                <PrimarySuffix />
        </LocalVariableDeclaration>
</BlockStatement>
```

In addition to local variable declarations, variables are also declared in the parameters of methods and as a field of a class. All three methods will have different syntax trees and need to be processed differently. After finding a variable declaration block statement such as this, by searching further up the tree until finding a class or method declaration the class and method name can be found.

The methods provided by the VariableStore class are: `add`, takes name, type, method name and class name of a variable and adds it to the store; `search`, takes type, method and class name of a variable and returns Vector of variable matching the description; `resolve`, takes name, method and class name of a variable and returns the type; `methodsInClass` takes the name of a class and returns a Vector of the methods available in the class.

4.6. Removal of extraneous information

In order to facilitate removal of extra information, primarily logging information, two methods are provided, `remove` and `removeAll`. The method named `remove` takes four arguments: a pointer to a section of the abstract syntax tree; the name of an object to remove; and the names of both the class and method to remove them from. The tree is then parsed to look for uses of the object. When a reference to the object is found, the block statement containing the call is removed using the `removeChild` method provided by the SimpleNode class.

In order to remove all uses of an object type, such as the Logger object, the `removeAll` method can be used. This method takes a single argument, a string representing the name of the object type to be removed. The `removeAll` method

then iterates through every class, calls the `methodsInClass` method from the VariableStore class to return all the methods in that class and then calls the `search` method from the VariableStore class to find what names are given to that type in each method. The `remove` method is then called with all of this information

4.7. Generating LySa processes

In order to generate a LySa process, several helper methods are needed. The main LySa process generation is performed in the `getLysaOutput` method. Conversion from the Java source to the LySa process requires searching through the Java input for uses of certain objects and methods.

lysaAppend(String toAppend) method

This method is used to append a String to the LySa process generated for the protocol. The LySa process for each principal is stored in a separate element of a Vector. The value of the element of the Vector being written in is determined by a global variable which is incremented after each class has been processed.

dumpLysa () method

This method outputs the LySa processes generated by calls to the `lysaAppend` method above. A ListIterator is created to iterate over the values of the Vector used to store each principal's LySa process. Parallel composition between the LySa processes is then added. Based on arguments given to Elyjah at the time of the running, the output will either be to the console or written to a lysa file.

findBSs(SimpleNode node, String prefix, String suffix) method

In the abstract syntax tree, a line of code is represented as a BlockStatement. This method returns a Vector of SimpleNodes each pointing to a BlockStatement and takes as arguments a pointer to a section of the abstract syntax tree and two strings, one for a prefix of a method call (which class or object the method belongs to) and a suffix (which method is being called).This method is used to return a list of all block statements containing a particular method call of a particular object. For example, this can be used to return abstract syntax representations of all calls to the `add` method of a particular Vector object in a block of code.

findDecryptionProperties(SimpleNode node) method

This method takes a SimpleNode representing the root of a method or some other block of code. This method searches for all calls to the decrypt method in this code block. It then extracts the key information from each of these method calls. The three pieces of information that need to be extracted are the key used, the name of the String being decrypted and the name of the Vector object the decrypted message is stored in. If crypto points are specified, then these will also be extracted at this point.

All of this information is stored in Vector objects. This means that if there are several `decrypt` method calls in the code block, all of them will be extracted and processed in turn.

seperateSwitchBlocks(SimpleNode node) method

This method will take a SimpleNode pointing to the root of a code block and return pointers to the separate switch blocks allowing each to be parsed independently. Pointers to the roots of each switch block are stored in a vector allowing them to be processed one by one. This is used to split the input file's processIncoming method into separate blocks, for which the LySa process for each process can be generated independently.

getNewExpressions(SimpleNode node) method

This method is called on each code block before generating the rest of the LySa process. The job of this method is to create LySa restriction processes. There are three different methods in the Java framework that must be converted into a LySa restriction process. This method looks for the `generateMessage`, `generateSharedKey` and `generateKeyPair` methods in a code block. When a `generateMessage` method is found, a LySa restriction process is appended using the variable name the message is bound to. If a key generation method is found then the generated key is resolved using a call to the `resolveKey` method, a LySa restriction process is then generated using the returned key name.

resolveKey(String keyName, SimpleNode codeBlock) methods

This method is used to resolve the name of a key given the name of the variable used to hold a key and a pointer to the root of the code block the variable is used in, In order to resolve a key, the variable name used to store the key must be converted to the string representation used to identify the key in the protocol. This is because in different methods, there is no guarantee that the developer will use the same variable name to store the same key. The name used to identify the key in the protocol can be retrieved from `registerKey` method calls or `get` method calls to the key store. For example, if the `resolveKey` method was called giving the name "key" and the surrounding block of code contained the method call given below, this method would return the String "KAS".

```
SecretKey key = (SecretKey) keys.get("KAS");
```

The string used to access the key is also appended with a + or – to signify the key is a public or private key respecitvely, if this is not already part of the string used to represent the key in the key store.

encryptionBlock(SimpleNode block, SimpleNode containingBlock) method

As encryption can occur at multiple places in a Java model of a protocol, it is most efficient to have a separate method to deal with generating a LySa encryption process. This method takes two SimpleNode objects as parameters, one pointing to a BlockStatement containing the `encrypt` method call, the other the surrounding block of code, either the `run` method or switch block. The arguments of the `encrypt` method reveal the Vector being encrypted and the key used to perform the encryption.

Firstly a String object which will later be the result of this method is created initialised to opening curly braces, "{". If the key used to encrypt the message is a PublicKey, then vertical bars must be appended this string. The contents of the Vector being encrypted can then be worked out through a call to the `findBSs` method using the Vector name and "add" as the method's arguments. This populates a Vector containing all lines of code containing a call to the Vector's `add` method. This Vector is used to create a ListIterator to process each `add` method call. If the item being added is a string literal, then this value is added to the return String. If the item added is a string variable then the name of the variable is added to the return String. If the item is a call to the `sendKey` method, then the `resolveKey` method is called on the argument of the `sendKey` method. The returned String from the `resolveKey` method is then added to the return String. If the item is a call to the `encrypt` method then the returned String from a recursive call to this method is added. If the key used is a PublicKey then another vertical bar is needed, either way a closing curly brace is added to the return String.

After these curly braces, the name of the key must be appended. To do this the `resolveKey` method must be applied to the variable used to store the key. After this the crypto point must be appended by examining the third and fourth arguments to the `encrypt` method.

findDecryptionProperties(SimpleNode node) method

This method is used to generate several Vector objects containing the key properties of `decrypt` method calls. This method scans a code block searching for calls to the `decrypt` method. For each method call found, the following properties need to be mined: the name of the string being decoded; the name of the Vector the decrypted message is stored in; and the name of the key used to decrypt the message. Where possible, the values from the crypto points will also be stored in separate vectors. In each Vector, the i-th position will hold a single property of the same call to the decrypt property.

getLysaOutput() method

There are two `getLysaOutput` methods in Elyjah, one without arguments, one that accepts a SimpleNode as an argument. The first is called in the Elyjah's `main` method after all the extraneous information has been removed. This method will then call the `getNewExpressions` method on the protocol set up class and then call the

second `getLysaOutput` method on each principal class in turn by providing the SimpleNode representing the root of the class as the method's argument.

getLysaOutput(SimpleNode classRoot) method

The first operation in each class is to identify the name of the variable used to contain the Network class; this is discovered with a call to the `search` method from the VariableStore class. Once this is known, the first task is to discover whether there is a `run` method in the class, if so this method is processed first. Firstly the `getNewExpressions` method is called on this method. The `findBSs` method is then used to retrieve a Vector of SimpleNodes, each representing a call to the `send` method of the local instance of the network class. A ListIterator is then created to work through this Vector, using a while loop to process each `send` method call in turn.

For each `send` method, firstly the opening chevron, '<', is added to the LySa process by calling Elyjah's `lysaAppend` method. The current class and the second argument of the send command are used to construct the first two parts of the LySa process. The third argument of the `send` method gives the name of the Vector being sent. Elyjah's `findBSs` method is then used to find all calls to the `add` method of this vector. The returned Vector is once again used to create a ListIterator to process each `add` method call. If the item being added is a string literal, then this value is added to the LySa process. If the item added is a string variable then the name of the variable is added to the LySa process. If the item is a call to the `sendKey` method, then Elyjah's `resolveKey` method is called on the `sendKey` argument. The returned String from the `resolveKey` method is then added to the LySa process. If the item is a call to the `encrypt` method then the returned String from Elyjah's `encryptionBlock` method is used. Finally the closing chevron and a full stop, '> .' is appended to the LySa process.

Once the `run` method has been processed, the same `processIncoming` method of the same class must be parsed. Firstly the method is split into separate switch blocks using Elyjah's `seperateSwitchBlocks` method. Once more a ListIterator is used to process each switch block in turn. This process involves several stages, firstly the receive message process is generated, and then any decryption processes, followed by any send processes.

Firstly, an opening parenthesis is appended using the `lysaAppend` method. In order to construct the LySa receive process, there are two stages. The first is to find all `check` method calls inside the switch block; this is done by calling the `findCheckBlockStatements` method. This method is similar to the `findBSs` method, but due to the different abstract syntax tree for the check block statements requires a slightly different method. The Vector generated from this method contains all the lines of code of the form

```
assert check(v.elementAt(0), "A");
```

This vector is used to create a ListIterator to run through each of these lines. From each one the string literal second argument of the check method is added to the LySa process, e.g A in the example above. It is possible for the second argument to be a call to the `encrypt` method, if this is the case then this is dealt with by the `encryptionBlock` method. When there are no more of these a semi-colon is appended and the variable bonding part of the receive process is generated. This is performed in a similar manner to the pattern-matching part of the receive process. This time a call to the `findAssignmentBlockStatements` method allows a ListIterator of the generated Vector to be processed. This method finds all lines of code of the form:

```
String msg = v.elementAt(2);
```

For each assignment like above, the name of the variable the incoming String is assigned to must be added to the LySa process. While doing this the number of the element of the received Vector is monitored. If at any point the element being accessed is not exactly one higher than the previous element, a warning message is output. Finally, a closing parenthesis and a full stop is appended to the LySa process.

After the receive message process has been completed a call is made to the `findDecryptionProperties` method giving the root of the switch block. The generated vectors should all have the same number of Strings in them, with the exception of the crypto point ones, which may have less. A ListIterator is then created using one of the main Vectors. For each call to `decrypt` append “decrypt”, then the name of the received string to be decoded, followed by “as {“ then use the same method as with the receive process to compose the pattern matching/ variable binding internals. Append closing parenthesis and name of key worked out from calling `resolveKey` with the variable name generated from the previous call to `findDecryptionProperties`. If the key used to encrypt the message is a PublicKey, then vertical bars must be also be appended to the LySa process inside the curly braces.

After any LySa decryption processes needed have been generated, any send processes are generated in the same manner as when processing the `run` method. Once this process has been completed for every principal class, the LySa process can be output to the console or a file by calling the `dumpLysa` method described above.

5. Testing and Evaluation

This section details the steps taken to ensure that Elyjah met the specification correctly. As the system was implemented in a pseudo extreme programming methodology many of the bugs were caught early in the development process. However towards the end of the project further steps were taken to ensure the correctness of the LySa output.

5.1. Test Suite

A test suite was designed that would allow for automated testing of Elyjah. This test suite would include input that covered four types of test data:

- Easy-to-compute input
- Typical input
- Boundary input
- Invalid input

Easy-to-compute input allows for regression testing of converting the most basic input that the Java framework would allow. These are very simple protocols such as one principal sending a message to another.

Typical input includes a number of real-world protocols such as the Needham-Schroeder protocol and the Wide Mouth Frog protocol.

Boundary input for Elyjah involves passing Java models of protocols that are not exactly as the specified input. It is expected that these inputs may not produce the correct LySa process; however the tests will check Elyjah's error handling capabilities.

Invalid input will involve invalid Java models, either programs with syntax errors, or Java programs that are not designed to be used with Elyjah in the first place. Again the test here is to see how Elyjah handles this erroneous input as opposed to whether the correct LySa process is generated.

All of the input files used in the test suite and the LySa output are included in the appendix of this report.

5.1.1. Easy-to-compute Input

There are five basic tests which test the most basic functionality of the Java framework and Elyjah itself. These tests serve as regression tests for both parts of the system. These tests test the ability of the Java framework to send and receive messages using both symmetric and asymmetric encryption.

TestEasySend.java

This file only implements one principal. This principal constructs a message with two parts and attempts to send it to a second principal that makes no attempt to process the incoming message. For this reason, no output is expected when run as a Java program, although Elyjah will accept it and return a correct LySa process. The LySa process this test should return is:

```
!(new msg) <A,B,A,msg>.0
|
!0
```

While as a whole this is not a valid protocol, the first line is the correct LySa process. The second line is a result of a second principal without an implemented `processIncoming` method.

TestEasySendReceive.java

This test is the simplest possible protocol that is a run able Java program. This file contains implementations of two principals. The protocol involves one principal sending a plain-text message to another. Elyjah should accept this file and return the following LySa process.

```
!(new msg) <A,B,A,msg>.0
|
!(A,B;source,message).0
```

When run as a Java program, the following output will be expected, with the current date and time in the appropriate places.

```
Date Time B processIncoming
INFO: A
Date Time B processIncoming
INFO: this is a message
```

TestEasySendEncrypt.java

This test, much like the first, does not produce any Java output. Just as the first test, an attempt is made to send a message to a second principal which makes no attempt to process the incoming message. However this message has been encrypted with a SecretKey. When this file is input to Elyjah, the following LySa process should be generated.

```
(new K) (
!(new msg) <A,B,A,{msg} : K [ at a dest {b} ] >.0
|
!0
)
```

TestEasySendReceiveEncrypt.java

This file is a continuation of the last test. This file has a fully implemented second principal that decrypts the message and prints both the encrypted and decrypted message using the Logger object. The Elyjah output using this file is the following LySa process.

```
(new K) (
!(new msg) <A,B,A,{msg} : K [ at a dest {b} ] >.0
|
!(A,B;source,msgToBeDecoded).
  decrypt msgToBeDecoded as {;message}:K [ at b orig {a} ]  in 0
)
```

The Java output of this file is the following Logger printouts, with the date and time where appropriate. The encrypted version of the message will also change with every run, although below is a typical string.

```
Date Time B processIncoming
INFO: A
Date Time B processIncoming
INFO: GvpN6kBrR1tfnARiZ8hOdXmd6PYlOwhn
Date Time B processIncoming
INFO: this is a message
```

TestEasySendReceivePublicEncrypt.java

The final simple test, is the most basic asymmetric encryption protocol possible. This protocol consists of two principals. One creates a public and private key pair, and sends the public key to the second principal. The second principal then uses this key to encrypt a message which it sends back. The first principal can then decrypt the message using the private key. The LySa process generated by Elyjah is as follows:

```
!(new +- K) <A,B,K+>.(B,A;msgToBeDecoded).
      decrypt msgToBeDecoded as {|;msg|}:K- [ at a orig {b} ]  in 0
|
!(A,B;K+).(new message) <B,A,{|message|} : K+ [ at b dest {a} ] >.0
```

When this Java program is run, the first protocol will print out the encrypted and decrypted message.

```
Date Time A processIncoming
INFO:
S33eIcQ2BIElRB7c7ELHdL9fFHGlHmwrjA5c4ZvB/CwG/Xe4qhV2eN93SaA
Mxt1oM/K1nMTF/VlcM6X9sEHOIOGio4K/CxyKhn550bX4JFYKokoNDOqIO
xLel7MN4IOByVOVJWLRTPweOSBSnlp9uknjax06B0iT8M7/c8rCDLE=
Date Time A processIncoming
INFO: this is the message
```

5.1.2. Typical Input

The typical input Elyjah will be used to process will be more complex protocols than the ones presented in the previous section. The following protocols have more than two principals and are well-known protocols covered in standard security courses.

WideMouthFrog.java

The Wide Mouth Frog protocol has already been well covered in the Background Information of this paper. This protocol has three principals, several different instances of symmetric encryption and allows two principals to securely communicate utilizing the third as a trusted third party server, The LySa process representing the WMF protocol that Elyjah should output is given below. The generated LySa process is equivalent to a manually composed LySa process for the WMF protocol that is given in [12].

```
(new KAS) (new KBS) (
!(new K) (new msg1) (new msg2) <A,S,A,{B,K} : KAS [ at a1 dest {s1} ] >.
        <A,B,{msg1,msg2} : K [ at a2 dest {b2} ] >.0
|
!(S,B;msgToBeDecoded).
        decrypt msgToBeDecoded as {A;KAB}:KBS [ at b1 orig {s2} ]  in
        (A,B;msgToBeDecoded2).
        decrypt msgToBeDecoded2 as {;msg1,msg2}:KAB [ at b2 orig {a2} ]  in 0
|
!(A,S,A;msgToBeDecoded).
        decrypt msgToBeDecoded as {B;keyRepresentation}:KAS [ at s1 orig {a1} ]
        in <S,B,{A,keyRepresentation} : KBS [ at s2 dest {b1} ] >.0
)
```

The Java output from running this protocol is given below. This represents the secure information sent from principal A to principal B encoded under a symmetric key that is sent via principal S.

```
Date Time B processIncoming
INFO: FIRST BIT OF MESSAGE
Date Time B processIncoming
INFO: SECOND BIT OF MESSAGE
```

WideMouthFrog2.java

In this second version of the Wide Mouth Frog protocol a deliberate error was introduced. In this version, the first message from principal A to principal S is completely unencrypted. This means that the key later used to allow A and B to communicate securely will be available to anyone. The LySa process will therefore be:

```
(new KAS) (new KBS) (
!(new K) (new msg1) (new msg2) <A,S,A,B,K>.
        <A,B,{msg1,msg2} : K [ at a2 dest {b2} ] >.0
|
!(S,B;msgToBeDecoded).
        decrypt msgToBeDecoded as {A;KAB}:KBS [ at b1 orig {s2} ]in
        (A,B;msgToBeDecoded2).
        decrypt msgToBeDecoded2 as {;msg1,msg2}:KAB [ at b2orig {a2} ]  in 0
|
!(A,S,A,B;keyRepresentation).
        <S,B,{A,keyRepresentation} : KBS [ at s2 dest {b1}] >.0
)
```

The Java output will be the same as the legitimate Wide Mouth Frog protocol mode.

NeedhamSchroeder.java

This file models the Needham Schroeder protocol. This protocol was invented by Roger Needham and Michael Schroeder in 1978. It allows individuals to prove their identity while preventing eavesdropping. In the Needham Schroeder protocol, it is required for one principal to make an amendment to a nonce provided by another principal. This amendment is usually a simple arithmetic operation; however it is impossible to model such a process in the LySa process calculus. Instead, all such operations have to be modelled by encryption. This is the reason behind principal A encrypting the nonce with a key SUCC before sending it to B, and principal B encrypting the same nonce when pattern matching the message from A. The key SUCC models an arithmetic operation such as successor. Performing the encryption on the same nonce in both principals models performing some arithmetic operation in both principals. The LySa process generated by Elyjah should thus be as follows.

```
(new KAS) (new KBS) (new SUCC) (
!(new nonceA) <A,S,A,B,nonceA>.(S,A;msgtoBeDecoded).
        decrypt msgtoBeDecoded as {nonceA,B;K,encryptedMsgToB}:KAS in
        <A,B,encryptedMsgToB>.(B,A;msgtoBeDecoded2).
        decrypt msgtoBeDecoded2 as {;nonceB}:K in
        (new msg1) (new msg2) <A,B,{{nonceB} :SUCC} : K>.
        <A,B,{msg1,msg2} : K>.0
|
!(A,B;msgToBeDecoded).
        decrypt msgToBeDecoded as {;KAB,source}:KBS in
        (new nonceB) <B,A,{nonceB} : KAB>.(A,B;msgToBeDecoded2).
        decrypt msgToBeDecoded2 as {{nonceB} : SUCC;}:KAB in
        (A,B;msgToBeDecoded3).
        decrypt msgToBeDecoded3 as {;msg1,msg2}:KAB in 0
|
!(A,S,A;destination,nonce).(new K) <S,A,{nonce,destination,K,{K,A} : KBS} : KAS>.0
)
```

The output from the logger object should be as below. This represents the secure communication decrypted at B.

Date Time B processIncoming
INFO: FIRST BIT OF MESSAGE
Date Time B processIncoming
INFO: SECOND BIT OF MESSAGE

5.1.3. Boundary Input

Boundary input for this system refers to a Java file that is a valid protocol implementation using the framework design but with deliberate flaws that may disrupt Elyjah's attempts to convert the model into a LySa process. There are three areas identified where this is possible. In an ideal implementation these flaws should not make a difference to the LySa process generated but appropriate error catching will be sufficient for this project.

Extraneous cast statements

Inserting a cast statement where it is not needed will not change the Java model's meaning but will change the abstract syntax tree of the line. An extraneous cast statement is one where the returned type is already the same as the required type but a cast is made to this type anyway, for example:

```
String message = (String) "this is a message";
```

This should not affect the generation of LySa processes but it is expected that Elyjah will not be able to cope due to the difference in the syntax tree.

Wrong ordering of statements

While the order of some statements will not make any difference, an area where they may make a difference is when constructing the receive and decrypt LySa processes. When constructing send and encrypt processes, Strings are added to the Vector in the order they are added, so changing the order of the add statements will change the order of the elements in the LySa process. However, when constructing decrypt and receive processes, it is possible to access the Vector object's elements out of order.

```
assert check(v.elementAt(1), "B");
assert check(v.elementAt(0), "S");
String msg = v.elementAt(2);
```

In the generated LySa process, the sender should be identified as S and the receiver B. However, if the order of the method calls is used to define the order of the LySa process then this order will be reversed. At the very least, there should be a warning to the developer that the statements are out of order.

Missing Variable Binding

If part of a received or decrypted message is not being checked in the pattern matching and is not used to generate a later message the developer may not bother to bind this part of the message to a variable. This will not affect the running of the Java model but will not produce a valid LySa process. While the LySa process will match the model, it will not be representative of the developer's idea. Additionally, if this part of the message is the final part received then there is no way for Elyjah to know the developer made a mistake.

In both of these final two test cases, the actual meaning of the input is ambiguous. It can never be determined if the developer meant to re-order the method calls or whether they made a mistake with the numbering. Equally, when it seems like a developer has missed out a number, they may have meant to add it later.

5.1.4. Invalid Input

The final type of test data that needs to be covered is erroneous input files. These are files which are badly constructed Java models, Java files not designed to be used with Elyjah or non java files. In all of these cases, Elyjah should return an error message indicating that the files are invalid or no valid LySa process could be mined from the file.

Badly constructed or non-Java file

In these cases, the parser should return an error message detailing the nature of the problem. Badly parsed files should be found during the construction of the abstract syntax tree. A parse error will take the form of the following message.

> Parse error at line 17, column 9. Encountered: net
> Java Parser Version 1.1: Encountered errors during parse.

This will be a regression test to make sure the parser is still functional. Any errors with invalid Java files should be caught before any attempt is made to extract a LySa process from the file.

Java files not designed to be processed by Elyjah

When no LySa process is found by Elyjah, instead of not outputting anything, an error message should be displayed saying that the file was not a valid protocol model.

5.2. Test suite results

This is the result of running the test files through Elyjah at the end of the implementation.

Input File	Test Result
TestEasySend.java	Correct LySa process generated - Pass
TestEasySendReceive.java	Correct LySa process generated - Pass
TestEasySendEncrypt.java	Correct LySa process generated - Pass
TestEasySendReceiveEncrypt.java	Correct LySa process generated - Pass
TestEasySendReceivePublicEncrypt.java	Correct LySa process generated - Pass
WideMouthFrog.java	Correct LySa process generated - Pass
WideMouthFrog2.java	Correct LySa process generated - Pass
NeedhamSchroeder.java	Correct LySa process generated - Pass
TestEasySendReceive.java with extraneous cast statements	No LySa process generated but correct error message displayed – Partial Fail
TestEasySendReceive.java with rearranged check statements	Wrong LySa process generated but correct error message displayed – Partial Fail
TestEasySendReceive.java with missing variable binding	LySa process generated according to Java model. No attempt to fix model.
TestEasySendReceive.java with missing semi-colon at end of line	No LySa process generated and correct error message displayed - Pass
TestEasySendReceive.class (Non-Java file)	No LySa process generated and correct error message displayed - Pass
ComClass.java	No LySa process generated and correct error message displayed - Pass

Elyjah consistently generated the correct LySa process for both the easy and typical inputs it was tested with. It also correctly dealt with invalid data and achieved a partial success on border line data by correctly generating warnings when the input was ambiguous or not as specified.

5.3. LySatool output of typical input

Once the Java model of a protocol has been converted into a LySa file it can be directly input into the LySatool. In the following section the results of the analysis by the LySatool have been summarised and explained.

WideMouthFrog.lysa

The LySatool renames many of the variables before analysing the protocol which accounts for the value names being different between the LySa file and the output. The LySatool output for the Wide Mouth Frog protocol is summarised below:

Values that may not be confidential
$\{LB, LK\}_{LKAS}$ [at a1 dest { s1 }], $\{LA, LkeyRepresentation\}_{LKBS}$ [at s2 dest { b1 }], $\{Lmsg1, Lmsg2\}_{LK}$ [at a2 dest { b2 }], n., m.+, m.-, S, B, A, $\{l., l.\}_{l.}$ [at CPDY]

Violation of authentication properties (ψ)
No violations possible

The first section details all the values that an attacker can read. This reveals that while an attacker can read the encrypted values of the messages, it can't decrypt the message and read the plaintext messages inside.

The second section simply confirms that it is not possible for an attacker to decrypt any messages nor can it create any messages which a principal can then be fooled into decrypting as part of a legitimate protocol.

WideMouthFrog2.lysa

With the deliberate error introduced in this protocol, it would be expected to receive a different result when using this file with the LySatool.

Values that may not be confidential
K, B, $\{LA, LkeyRepresentation\}_{LKBS}$ [at s2 dest { b1 }], $\{Lmsg1, Lmsg2\}_{LK}$ [at a2 dest { b2 }], n•, m•+, m•-, S, A, keyRepresentation, $\{l•, l•\}_{l•}$ [at CPDY], msg1, msg2

Violation of authentication properties (ψ)
(CPDY, b2), (a2, CPDY)

Here we can see that by making this one change the protocol is completely compromised. The first section reveals that the message that A is attempting to send to B is not confidential. Additionally, an attacker can pose as A and send B a message of its own.

NeedhamSchroeder.lysa

Values that may not be confidential
nonceA, B, $\{l•, l•, LK, L71\}_{LKAS}$, $\{LK, LA\}_{LKBS}$, $\{LnonceB\}_{LKAB}$, $\{L29\}_{LK}$, $\{Lmsg1, Lmsg2\}_{LK}$, n•, m•+, m•-, S, A, $\{l•, l•\}_{l•}$ [at CPDY], $\{l•, l•, l•, l•\}_{l•}$ [at CPDY], $\{l•\}_{l•}$ [at CPDY]

Violation of authentication properties (ψ)
No violations possible

The analysis of this protocol is very similar to that of the Wide Mouth Frog protocol. The first section reveals that an attacker can read any of the decrypted messages but can not read the contents. The second section reveals that, like the WMF protocol, an attacker can not decrypt any of the messages or create their own that a principal will decrypt.

6. Conclusion

6.1. Shortcomings and Solutions

Using the test suite revealed that Elyjah works in all situations where the Java input is exactly as specified. However in cases where the Java input was not exactly as expected, possible shortcomings were identified. The first such shortcoming occurs when the order of the `elementAt` method calls to a Vector object are not in ascending numerical order. In these instances, the Java model will work correctly but the LySa process is not correctly generated. The following table shows two Java code blocks that produce different LySa processes but running the code will produce the same result.

`assert check(v.elementAt(0), "S");`
`assert check(v.elementAt(1), "B");`
`String msg = v.elementAt(2);`
 	
Currently produces LySa Process:
(S, B; msg)
	`assert check(v.elementAt(1), "B");`
	`assert check(v.elementAt(0), "S");`
	`String msg = v.elementAt(2);`
	
	Currently produces LySa Process:
	(B, S; msg)

While Elyjah will produce an error code at this point, it would be possible to produce a correct LySa process. This could be achieved by storing the String to be added to the LySa process in a temporary Vector at the position it was removed from, then creating a ListIterator to add of the contents of this temporary Vector, when all the elements have been filled.

A second area where Elyjah can be improved is when a developer misses out a variable binding in a receive or decrypt process. If this variable is not used in the construction of any other message, then it does not matter what name is assigned to that part of the message. The following two Java blocks should be able to produce the same LySa process providing the variable msg2 is not used again.

`assert check(v.elementAt(0), "S");`
`assert check(v.elementAt(1), "B");`
`String msg = v.elementAt(2);`
`String msg2 = v.elementAt(3);`
`String msg3 = v.elementAt(4);`
 	
Currently produces LySa Process:
(S, B; msg, msg2)
	`assert check(v.elementAt(0), "S");`
	`assert check(v.elementAt(1), "B");`
	`String msg = v.elementAt(2);`
	`String msg3 = v.elementAt(4);`
	
	Currently produces LySa Process:
	(S, B; msg2)

Currently Elyjah produces the same warning generated with out-of-order Vector `elementAt` methods. However, if possible, the missing variable in the above example could be filled. This could be achieved by adding a randomly named variable in place of the missing one.

However, in both of these cases, it may be that the developer made an honest mistake and mistyped the number of the `elementAt` method's argument. Attempting to second guess the developer's model in these cases is likely to produce a LySa model that does not represent the developer's vision for the

protocol. For this reason, Elyjah makes no attempt to implement the proposed solutions above and restricts itself to providing warnings.

6.2. Possible Future Work

Even though the project goals have been met, there are many ideas that could be used to extend Elyjah in order to increase its functionality. One idea would be an addition which allows a graphical simulation of the Java model of the protocol. This allows the user's protocol to be represented pictorially such that a user can see messages passed between principals. There are a number of methods of representing protocols graphically. One option would be to represent the principals as nodes of a graph with messages travelling along edges between the nodes. It should be possible to observe the contents of a message as the message is being transmitted. This would also be useful as an educational tool. A similar display idea is used in the Framework Animations of Distributed Algorithms project maintained by the Centre for Research in IT and Education from Trinity College Dublin's computer science department. This allows a developer to observe a simulation of a distributed algorithm with messages being passed between different principals. Another option would be to display the protocol as an animated sequence diagram. This allows the full protocol to be seen on the screen at the same time while still showing the timing.

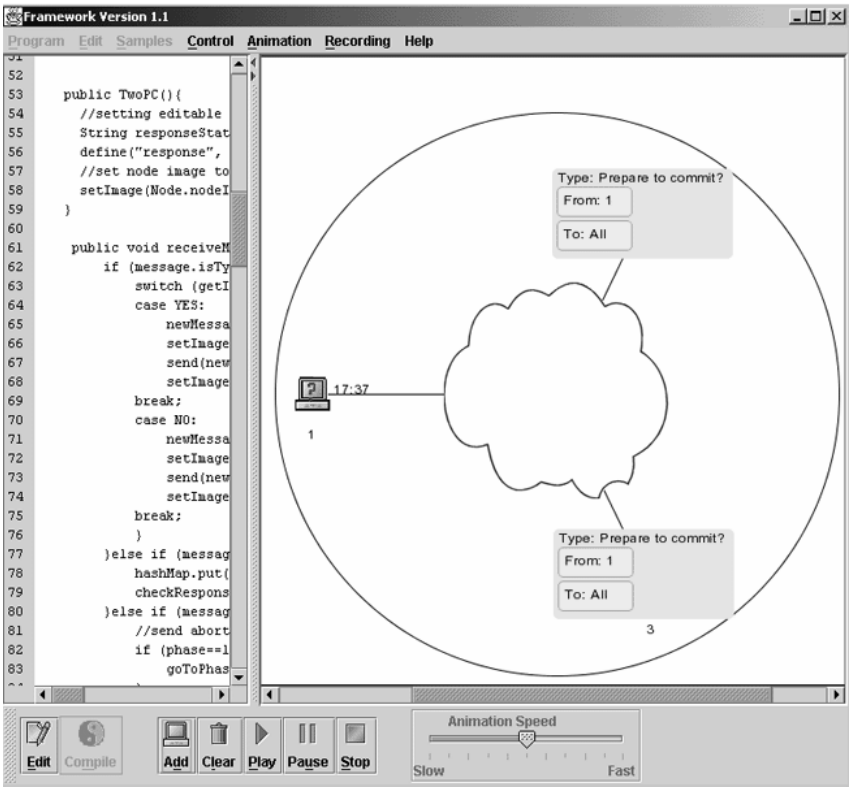

Figure 6 – Screen capture of the Framework Animations of Distributed Algorithms from https://www.cs.tcd.ie/Fionnuala.ODonnell/Framework/index.htm

Although challenging, it may also be possible to show the results of the LySatool in this pictorial representation. The difficulty here is that an attack is often a many stage process and the point of attack is often separated from the point where an error in the protocol is made. However, using the crypto points it would be possible to illustrate where an attacker can violate the authentication properties.

Another option for further development would be to extend Elyjah to output Meta LySa. Meta LySa is an extension of LySa which adds indexed constructs to all names, variables, and crypto-points. This allows for a concise representation of multiple copies of processes. Meta LySa can be converted to LySa through syntactic expansion a process automatically performed by the LySatool. Analysis of a Meta LySa process may reveal additional flaws in a protocol that analysis of a single copy of a protocol does not reveal. This is because an attacker may use messages from one instance of a protocol against a separate instance.

6.3. General Conclusion

The original aim of this project was to create a software tool to convert Java implementations of communication protocols into a process calculus which can then be analysed. Based on the evidence presented in the Section 5, Elyjah clearly meets this goal. Using Elyjah, developers can for the first time translate a working model of a protocol into LySa. Developers can use the system to help them implement protocols securely as well as learn more about communication protocols and the attacks that can be performed on them.

Through this project, a framework for easily modelling protocols in the Java language has also been developed. This framework helps developers to implement protocols by providing most of the required functionality while providing the flexibility to allow any protocol to be modelled.

There are no major flaws in the system although it can be improved to increase the robustness. Elyjah provides a strong foundation for future work. Both the Java framework and Elyjah itself can be extended to provide more functionality. I believe further development on Elyjah would be worthwhile. Over the past few years static analysis has been increasingly used in a range of fields with ever growing success. Interest in security is equally high at the moment. Over the course of this project I have come to understand why more developers do not use these formal techniques. This is mainly due to the lack of literature available regarding LySa targeted towards complete novices of process algebras. With such little information available for ordinary developers, further development of tools like Elyjah could really help to introduce more people to these formal methods.

Bibliography

[1] M. Buchholtz *User's Guide for the LySatool version 2.01*
Retreived August 2005 from:
http://www2.imm.dtu.dk/cs_LySa/lysatool/lysatool-2.01.pdf

[2] C. Bodei, M. Buchholtz, P. Degano, M. Curti, C. Priami, F. Nielson, H. Riis Nielson: *Performance Evaluation of Security Protocols Specified in LySa* Proceedings of the 2nd Workshop on Quantitative Aspects of Programming Languages (QAPL 04), ENTCS vol. 112, p 167-189, 2005.

[3] D. Dolev, & A. C. Yao, *On the Security of Public Key Protocols*
IEEE Transactions on Information Theory Vol IT-29, No2. March 1983 pp 198-208

[4] C. Bodei, M. Buchholtz, P. Degano, F. Nielson. & F. Riis Nielson *Automatic Validation of Protocol Narration*
Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW'03) pp 126-140. IEEE Computer Society Press. 2003.

[5] M. Buchholtz, C. Montangero, L. Perrone & S. Semprini *For-LySa: UML for Authentication Analysis*
Global Computing: IST/FET International Workshop, GC 2004, LNCS vol. 3267, p. 93-106, Springer Verlag, 2005

[6] C. Bodei, M. Buchholtz, P. Degano, F. Nielson & H. Riis Nielson *Static Validation of Security Protocols*
Journal of Computer Security.

[7] M. Buchholts, S. Gilmore, J. Hilston & F. Nielson *Securing statically-verified communications protocols against timing attacks*
Proceedings of First International Workshop on Practical Applications of Stochastic Modelling (PASM 04)

[8] L. Gong *Inside Java 2 Platofrm Security: Architecture, API Design and Implementation*
Addison-Wesley (ISBN: 0-201-31000-7)

[9] A. Appel *Modern compiler implementation in Java / 2nd ed.*
Cambridge : Cambridge University Press, 2002

[10] C. Montangero, L. Perrone, and S. Semprini *For-LySa: UML for Authentication Analysis*
DEGAS IST-2001-32072

[11] C. Bodei, M. Buchholtz, M. Curti, P. Degano F. Nielson, H. Riis Nielson, C. Priami *On Evaluating the Performance of Security Protocols*

[12] S. Gilmore *Securing statically-verified communications protocols against timing attacks*
Modelling, Methods and Tools November 14 2005 lecture note
http://homepages.inf.ed.ac.uk/stg/teaching/mmt/slides/pepa_lysa.pdf

Appendix

Java Framework method headers

ComClass Methods

```
public abstract void processIncoming(Vector<String> v);
public void shareKey(Key key, String name)
public void registerKey(Key key, String name)
public KeyPair generateKeyPair(long userseed)
public SecretKey generateSharedKey()
public String generateMessage(String message)
public boolean check(Object a, Object b)
public String sendKey(Key key)
public SecretKey receiveSecretKey(String str)
public PublicKey receivePublicKey(String str)
public String encrypt(Vector v, Key key, String at, Vector<String> dest)
public Vector decrypt(String str, Key key, String at, Vector<String> orig)
public String encrypt(Vector v, Key key, String at, String dest)
public Vector decrypt(String str, Key key, String at, String orig)
public String encrypt(Vector v, Key key)
public Vector decrypt(String str, Key key)
```

KeyGenerationClass Methods

```
public static KeyPair generateKeyPair(long userseed)
public static SecretKey generateSharedKey()
```

Network Class Methods

```
public Network()
public void send(ComClass source, String dest, Vector tuple)
public void register(String name, ComClass comClass)
public void shareKey(Key key, String name)
```

Test Files

In the following pages are the Java files used in Section 5 to test Elyjah. After each file is the LySa file generated from this file. As well as these test cases the MultipleSendReceive,java example referenced in Section 2.2.4 is included here.

Elyjah input (MultipleSendReceive.java)

```
/*
 * MultipleSendReceive.java
 *
 * Created on 16 November 2005, 11:13
 *
 */
import java.util.Vector;
import javax.crypto.*;
/**
 *
 * @author Nicholas O"Shea
 */
public class MultipleSendReceive extends KeyGenerationClass{
    public static void main(String[] args) {
        Network net = new Network();
        A a = new A(net);
        net.register("A", a);
        B b = new B(net);
        net.register("B", b);
        a.start();
        b.start();
    }
}

class A extends ComClass {
    Network net;

    public A (Network net){
        this.net = net;
    }

    public void  run(){
        Vector v = new Vector();
        v.add("password");
        String msg = generateMessage("Message1");
        v.add(msg);
        net.send(this, "B", v);
    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0 :
                assert check(v.elementAt(0), "B");
                assert check(v.elementAt(1), "A");
                assert check(v.elementAt(2), "password2");
                String msg2 = v.elementAt(3);
                Vector v2 = new Vector();
                theLogger.info(msg2);
                v2.add("password3");
                String msg3 = generateMessage("message 3");
```

```
                v2.add(msg3);
                net.send(this, "B", v2);
                break;
            }
        }

}
class B extends ComClass {
    Network net;

    public B (Network net){
        this.net = net;
    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0:
                receivedNum++;
                // check first
                assert check(v.elementAt(0), "A");
                // check second
                assert check(v.elementAt(1), "B");
                // check third
                assert check(v.elementAt(2), "password");
                // assign fourth
                String msg =  v.elementAt(3);
                theLogger.info(msg);
                Vector v2 = new Vector();
                v2.add("password2");
                String msg2 = generateMessage("Message2");
                v2.add(msg2);
                net.send(this, "A", v2);
                break;
            case 1:
                assert check(v.elementAt(0), "A");
                assert check(v.elementAt(1), "B");
                assert check(v.elementAt(2), "password3");
                String msg3 = v.elementAt(3);
                theLogger.info(msg3);
               break;
            }

    }
}
```

Elyjah input (TestEasySend.java)

```
/*
 * TestEasySend.java
 *
 * Created on 16 November 2005, 11:13
 *
 */
import java.util.Vector;
import javax.crypto.*;
/**
 *
 * @author Nicholas O"Shea
 */
public class TestEasySend extends KeyGenerationClass {
    public static void main(String[] args) {
        Network net = new Network();
        A a = new A(net);
        net.register("A", a);

        B b = new B(net);
        net.register("B", b);

        b.start();
        a.start();
    }
}

class A extends ComClass {
    Network net;


    public A (Network net){
        this.net = net;
    }

    public void  run(){
        Vector v = new Vector();
        v.add("A");
        String msg = generateMessage("this is a message");
        v.add(msg);
        net.send(this, "B", v);



    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0 :
        }
    }
```

```
}
class B extends ComClass {
    Network net;


    public B (Network net){
        this.net = net;
    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0 :
        }
    }

}
```

Elyjah output (TestEasySend.lysa)

```
!(new msg) <A,B,A,msg>.0
|
!0
```

Elyjah input (TestEasySendReceive.java)

```
/*
 * TestEasySendReceive.java
 *
 * Created on 16 November 2005, 11:13
 *
 */
import java.util.Vector;
import javax.crypto.*;
/**
 *
 * @author Nicholas O"Shea
 */
public class TestEasySendReceive extends KeyGenerationClass {
    public static void main(String[] args) {
        Network net = new Network();
        A a = new A(net);
        net.register("A", a);
        B b = new B(net);
        net.register("B", b);

        b.start();
        a.start();
    }
}

class A extends ComClass {
    Network net;


    public A (Network net){
        this.net = net;
    }

    public void  run(){
        Vector v = new Vector();
        v.add("A");
        String msg = generateMessage("this is a message");
        v.add(msg);
        net.send(this, "B", v);



    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0 :
        }
    }

}
```

```
class B extends ComClass {
    Network net;


    public B (Network net){
        this.net = net;
    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0 :
                assert (check(v.elementAt(0), "A"));
                assert (check(v.elementAt(1), "B"));
                String source = v.elementAt(2);
                String message = v.elementAt(3);

                theLogger.info(source);
                theLogger.info(message);
        }
    }

}
```

Elyjah output (TestEasySendReceive.lysa)

```
!(new msg) <A,B,A,msg>.0
|
!(A,B;source,message).0
```

Elyjah input (TestEasySendEncrypt.java)

```
/*
 * TestEasySendEncrypt.java
 *
 * Created on 16 November 2005, 11:13
 *
 */
import java.util.Vector;
import javax.crypto.*;
/**
 *
 * @author Nicholas O"Shea
 */
public class TestEasySendEncrypt extends KeyGenerationClass {
    public static void main(String[] args) {
        Network net = new Network();
        A a = new A(net);
        net.register("A", a);
        B b = new B(net);
        net.register("B", b);

        SecretKey key = generateSharedKey();
        net.shareKey(key, "K");

        b.start();
        a.start();
    }
}

class A extends ComClass {
    Network net;


    public A (Network net){
        this.net = net;
    }

    public void  run(){
        Vector v = new Vector();
        v.add("A");
        String msg = generateMessage("this is a message");
        Vector vEnc = new Vector();
        vEnc.add(msg);
        SecretKey key = (SecretKey) keys.get("K");
        v.add(encrypt(vEnc, key, "a", "b"));
        net.send(this, "B", v);



    }

    public void processIncoming(Vector<String> v){
```

```
            switch (receivedNum) {
                case 0 :
            }
        }

}
class B extends ComClass {
        Network net;


        public B (Network net){
            this.net = net;
        }

        public void processIncoming(Vector<String> v){
            switch (receivedNum) {

            }
        }

}
```

Elyjah output (TestEasySendEncrypt.lysa)

```
(new K) (
!(new msg) <A,B,A,{msg} : K [ at a dest {b} ] >.0
|
!0
)
```

Elyjah input (TestEasySendReceiveEncrypt.java)

```
/*
 * TestEasySendReceiveEncrypt.java
 *
 * Created on 16 November 2005, 11:13
 *
 */
import java.util.Vector;
import javax.crypto.*;
/**
 *
 * @author Nicholas O"Shea
 */
public class TestEasySendReceiveEncrypt extends KeyGenerationClass {
    public static void main(String[] args) {
        Network net = new Network();
        A a = new A(net);
        net.register("A", a);
        B b = new B(net);
        net.register("B", b);

        SecretKey key = generateSharedKey();
        net.shareKey(key, "K");

        b.start();
        a.start();
    }
}

class A extends ComClass {
    Network net;


    public A (Network net){
        this.net = net;
    }

    public void  run(){
        Vector v = new Vector();
        v.add("A");
        String msg = generateMessage("this is a message");
        Vector vEnc = new Vector();
        vEnc.add(msg);
        SecretKey key = (SecretKey) keys.get("K");
        v.add(encrypt(vEnc, key, "a", "b"));
        net.send(this, "B", v);



    }

    public void processIncoming(Vector<String> v){
```

```
        switch (receivedNum) {
            case 0 :
        }
    }

}
class B extends ComClass {
    Network net;


    public B (Network net){
        this.net = net;
    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0 :
                assert (check(v.elementAt(0), "A"));
                assert (check(v.elementAt(1), "B"));
                String source = v.elementAt(2);
                String msgToBeDecoded = v.elementAt(3);
                SecretKey key = (SecretKey) keys.get("K");
                Vector<String> decode = decrypt(msgToBeDecoded, key, "b",
"a");
                String message = decode.elementAt(0);

                theLogger.info(source);
                theLogger.info(msgToBeDecoded);
                theLogger.info(message);
        }
    }

}
```

Elyjah output (TestEasySendReceiveEncrypt.lysa(

```
(new K) (
!(new msg) <A,B,A,{msg} : K [ at a dest {b} ] >.0
|
!(A,B;source,msgToBeDecoded). decrypt msgToBeDecoded as {;message}:K [
at b orig {a} ]  in 0
)
```

Elyjah input (TestEasySendReceivePublicEncrypt.java)

```
/*
 * TestEasySendReceivePublicEncrypt.java
 *
 * Created on 16 November 2005, 11:13
 *
 */
import java.security.KeyPair;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.util.Vector;
import javax.crypto.*;
/**
 *
 * @author Nicholas O"Shea
 */
public class TestEasySendReceivePublicEncrypt extends KeyGenerationClass {
    public static void main(String[] args) {
        Network net = new Network();
        A a = new A(net);
        net.register("A", a);
        B b = new B(net);
        net.register("B", b);

        b.start();
        a.start();
    }
}

class A extends ComClass {
    Network net;


    public A (Network net){
        this.net = net;
    }

    public void  run(){
        Vector v = new Vector();
        KeyPair keyPair = generateKeyPair(1024);
        PublicKey keyPublic = keyPair.getPublic();
        PrivateKey keyPrivate = keyPair.getPrivate();
        registerKey(keyPublic, "K+");
        registerKey(keyPrivate, "K-");
        v.add(sendKey(keyPublic));


        net.send(this, "B", v);



    }
```

```
    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0 :
                assert (check(v.elementAt(0), "B"));
                assert (check(v.elementAt(1), "A"));
                String msgToBeDecoded = v.elementAt(2);
                PrivateKey key = (PrivateKey) keys.get("K-");
                Vector<String> decode = decrypt(msgToBeDecoded, key, "a",
"b");
                String msg = decode.elementAt(0);

                theLogger.info(msgToBeDecoded);
                theLogger.info(msg);
        }
    }

}
class B extends ComClass {
    Network net;


    public B (Network net){
        this.net = net;
    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0 :
                assert (check(v.elementAt(0), "A"));
                assert (check(v.elementAt(1), "B"));

                PublicKey key = receivePublicKey(v.elementAt(2));
                registerKey(key, "K");

                Vector vMsg = new Vector();
                Vector vEnc = new Vector();
                String message = generateMessage("this is the message");
                vEnc.add(message);
                vMsg.add(encrypt(vEnc, key, "b", "a"));
                net.send(this, "A", vMsg);
           }
    }

}
```

Elyjah output (TestEasySendReceivePublicEncrypt.lysa)

```
!(new +- K-) <A,B,K+>.(B,A;msgToBeDecoded). decrypt msgToBeDecoded as
{|;msg|}:K- [ at a orig {b} ]  in 0
|
!(A,B;K+).(new message) <B,A,{|message|} : K+ [ at b dest {a} ] >.0
```

Elyjah input (WideMouthFrog.java)

```
/*
 * WideMouthFrog.java
 *
 * Created on 16 November 2005, 11:13
 *
 */
import java.util.Vector;
import javax.crypto.*;
/**
 *
 * @author Nicholas O"Shea
 */
public class WideMouthFrog extends KeyGenerationClass {
    public static void main(String[] args) {
        Network net = new Network();
        A a = new A(net);
        net.register("A", a);
        B b = new B(net);
        net.register("B", b);
        S s = new S(net);
        net.register("S", s);



        SecretKey keyA = generateSharedKey();
        SecretKey keyB = generateSharedKey();

        a.shareKey(keyA, "KAS");
        s.shareKey(keyA, "KAS");
        b.shareKey(keyB, "KBS");
        s.shareKey(keyB, "KBS");

        a.start();
        b.start();
        s.start();
    }
}

class A extends ComClass {
    Network net;


    public A (Network net){
        this.net = net;
    }

    public void  run(){
        Vector v = new Vector();
        v.add("A");
        Vector vEncoded = new Vector();
        vEncoded.add("B");
```

```
            SecretKey keyAB = generateSharedKey();
            SecretKey keyAS = (SecretKey) keys.get("KAS");
            registerKey(keyAB, "K");
            vEncoded.add(sendKey(keyAB));
            v.add(encrypt(vEncoded, keyAS, "a1", "s1"));
            net.send(this, "S", v);

            // Message to B
            Vector vB = new Vector();
            Vector vBEncoded = new Vector();
            String msg1 = generateMessage("FIRST BIT OF MESSAGE");
            String msg2 = generateMessage("SECOND BIT OF MESSAGE");
            vBEncoded.add(msg1);
            vBEncoded.add(msg2);
            vB.add(encrypt(vBEncoded, keyAB, "a2", "b2"));
            net.send(this, "B", vB);

        }

        public void processIncoming(Vector v){
            switch (receivedNum) {
                case 0 :
            }
        }

}
class B extends ComClass {
        Network e;

        public B (Network e){
            this.e = e;
        }

        public void processIncoming(Vector<String> v){
            switch (receivedNum) {
                case 0:
                    receivedNum++;
                    // check first
                    assert check(v.elementAt(0), "S");
                    // check second
                    assert check(v.elementAt(1), "B");
                    // check third
                    // assign fourth
                    SecretKey key = (SecretKey) keys.get("KBS");
                    String msgToBeDecoded = v.elementAt(2);
                    Vector<String> decode = decrypt(msgToBeDecoded, key,
"b1", "s2");


                    assert check(decode.elementAt(0), "A");
                    //String keyRepresentation = decode.elementAt(1);

                    SecretKey keyAB = receiveSecretKey(decode.elementAt(1));
```

```
                registerKey(keyAB, "KAB");

                break;
            case 1:
                receivedNum++;
                // check first
                assert check(v.elementAt(0), "A");
                // check second
                assert check(v.elementAt(1), "B");
                String msgToBeDecoded2 = v.elementAt(2);
                SecretKey key2 = (SecretKey)keys.get("KAB");
                Vector<String> decode2 = decrypt(msgToBeDecoded2, key2,
"b2", "a2");

                String msg1 = decode2.elementAt(0);
                String msg2 = decode2.elementAt(1);
                theLogger.info(msg1);
                theLogger.info(msg2);
        }

    }
}
class S extends ComClass {
    Network e;

    public S (Network e){
        this.e = e;
    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0:
                receivedNum++;
                // check first
                assert check(v.elementAt(0), "A");
                // check second
                assert check(v.elementAt(1), "S");
                assert check(v.elementAt(2), "A");
                SecretKey key = (SecretKey) keys.get("KAS");
                String msgToBeDecoded = v.elementAt(3);
                Vector<String> decode = decrypt(msgToBeDecoded, key,
"s1", "a1");

                assert check(decode.elementAt(0), "B");
                //SecretKey keyAB = (SecretKey) decode.elementAt(1);
                String keyRepresentation = decode.elementAt(1);

                Vector v2 = new Vector();
                v2.add("A");
                v2.add(keyRepresentation);
                Vector v3 = new Vector();
                SecretKey key2 = (SecretKey) keys.get("KBS");
                v3.add(encrypt(v2, key2, "s2", "b1"));
                e.send(this, "B", v3);
```

```
                    break;
            }
        }
}
```

Elyjah output (WideMouthFrog.lysa)

```
(new KAS) (new KBS) (
!(new K) (new msg1) (new msg2) <A,S,A,{B,K} : KAS [ at a1 dest {s1} ]
>.<A,B,{msg1,msg2} : K [ at a2 dest {b2} ] >.0
|
!(S,B;msgToBeDecoded). decrypt msgToBeDecoded as {A;KAB}:KBS [ at b1
orig {s2} ]  in (A,B;msgToBeDecoded2). decrypt msgToBeDecoded2 as
{;msg1,msg2}:KAB [ at b2 orig {a2} ]  in 0
|
!(A,S,A;msgToBeDecoded). decrypt msgToBeDecoded as
{B;keyRepresentation}:KAS [ at s1 orig {a1} ]  in
<S,B,{A,keyRepresentation} : KBS [ at s2 dest {b1} ] >.0
)
```

Elyjah input (WideMouthFrog2.java)

```
/*
 * WideMouthFrog2.java
 *
 * Created on 16 November 2005, 11:13
 *
 */
import java.util.Vector;
import javax.crypto.*;
/**
 *
 * @author Nicholas O'Shea
 */
public class WideMouthedFrog2 extends KeyGenerationClass {
    public static void main(String[] args) {
        Network net = new Network();
        A a = new A(net);
        net.register("A", a);
        B b = new B(net);
        net.register("B", b);
        S s = new S(net);
        net.register("S", s);



        SecretKey keyA = generateSharedKey();
        SecretKey keyB = generateSharedKey();

        net.shareKey(keyA, "KAS");
        net.shareKey(keyB, "KBS");

        a.start();
        b.start();
        s.start();
    }
}

class A extends ComClass {
    Network net;


    public A (Network net){
        this.net = net;
    }

    public void  run(){
        Vector v = new Vector();
        v.add("A");
        Vector vEncoded = new Vector();
        //vEncoded.add("B");
        v.add("B");
        SecretKey keyAB = generateSharedKey();
```

```
        SecretKey keyAS = (SecretKey) keys.get("KAS");
        registerKey(keyAB, "K");
        v.add(sendKey(keyAB));
        net.send(this, "S", v);

        // Message to B
        Vector vB = new Vector();
        Vector vBEncoded = new Vector();
        String msg1 = generateMessage("FIRST BIT OF MESSAGE");
        String msg2 = generateMessage("SECOND BIT OF MESSAGE");
        vBEncoded.add(msg1);
        vBEncoded.add(msg2);
        vB.add(encrypt(vBEncoded, keyAB, "a2", "b2"));
        net.send(this, "B", vB);

    }

    public void processIncoming(Vector v){
        switch (receivedNum) {
            case 0 :
        }
    }

}
class B extends ComClass {
    Network e;

    public B (Network e){
        this.e = e;
    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0:
                receivedNum++;
                // check first
                assert check(v.elementAt(0), "S");
                // check second
                assert check(v.elementAt(1), "B");
                // check third
                // assign fourth
                SecretKey key = (SecretKey) keys.get("KBS");
                String msgToBeDecoded = v.elementAt(2);
                Vector<String> decode = decrypt(msgToBeDecoded, key,
"b1", "s2");


                assert check(decode.elementAt(0), "A");
                //String keyRepresentation = decode.elementAt(1);

                SecretKey keyAB = receiveSecretKey(decode.elementAt(1));

                registerKey(keyAB, "KAB");
```

```
                    break;
               case 1:
                    receivedNum++;
                    // check first
                    assert check(v.elementAt(0), "A");
                    // check second
                    assert check(v.elementAt(1), "B");
                    String msgToBeDecoded2 = v.elementAt(2);
                    SecretKey key2 = (SecretKey)keys.get("KAB");
                    Vector<String> decode2 = decrypt(msgToBeDecoded2, key2,
"b2", "a2");

                    String msg1 = decode2.elementAt(0);
                    String msg2 = decode2.elementAt(1);
                    theLogger.info(msg1);
                    theLogger.info(msg2);
          }

     }
}
class S extends ComClass {
     Network e;

     public S (Network e){
          this.e = e;
     }

     public void processIncoming(Vector<String> v){
          switch (receivedNum) {
               case 0:
                    receivedNum++;
                    // check first
                    assert check(v.elementAt(0), "A");
                    // check second
                    assert check(v.elementAt(1), "S");
                    assert check(v.elementAt(2), "A");
                    SecretKey key = (SecretKey) keys.get("KAS");
                    //String msgToBeDecoded = v.elementAt(3);
                    //Vector<String> decode = decrypt(msgToBeDecoded, key,
"s1", "a1");
                    assert check(v.elementAt(3), "B");

                    //assert check(decode.elementAt(0), "B");
                    //SecretKey keyAB = (SecretKey) decode.elementAt(1);
                    //String keyRepresentation = decode.elementAt(1);
                    String keyRepresentation = v.elementAt(4);

                    Vector v2 = new Vector();
                    v2.add("A");
                    v2.add(keyRepresentation);
                    Vector v3 = new Vector();
                    SecretKey key2 = (SecretKey) keys.get("KBS");
                    v3.add(encrypt(v2, key2, "s2", "b1"));
                    e.send(this, "B", v3);
```

```
                    break;
            }
        }
}
```

Elyjah output (WideMouthFrog2.lysa)

```
(new KAS) (new KBS) (
!(new K) (new msg1) (new msg2) <A,S,A,B,K>.<A,B,{msg1,msg2} : K [ at a2
dest {b2} ] >.0
|
!(S,B;msgToBeDecoded). decrypt msgToBeDecoded as {A;KAB}:KBS [ at b1
orig {s2} ]  in (A,B;msgToBeDecoded2). decrypt msgToBeDecoded2 as
{;msg1,msg2}:KAB [ at b2 orig {a2} ]  in 0
|
!(A,S,A,B;keyRepresentation).<S,B,{A,keyRepresentation} : KBS [ at s2
dest {b1} ] >.0
)
```

Elyjah input (NeedhamSchroeder.java)

```
/*
 * NeedhamSchroeder.java
 *
 * Created on 16 November 2005, 11:13
 *
 */
import java.util.Vector;
import javax.crypto.*;
/**
 *
 * @author Nicholas O"Shea
 */
public class NeedhamSchroeder extends KeyGenerationClass {
    public static void main(String[] args) {
        Network net = new Network();
        A a = new A(net);
        net.register("A", a);
        B b = new B(net);
        net.register("B", b);
        S s = new S(net);
        net.register("S", s);



        SecretKey keyA = generateSharedKey();
        SecretKey keyB = generateSharedKey();
        SecretKey succ = generateSharedKey();

        a.shareKey(keyA, "KAS");
        s.shareKey(keyA, "KAS");

        b.shareKey(keyB, "KBS");
        s.shareKey(keyB, "KBS");

        net.shareKey(succ, "SUCC");

        a.start();
        b.start();
        s.start();
    }
}

class A extends ComClass {
    Network net;
    String nonceA;


    public A (Network net){
        this.net = net;
    }
```

```
    public void  run(){
        Vector v = new Vector();
        v.add("A");
        v.add("B");
        String nonceA = generateMessage("nonce");
        v.add(nonceA);
        net.send(this, "S", v);


    }

    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0 :
                receivedNum++;
                // check first
                assert check(v.elementAt(0), "S");
                // check second
                assert check(v.elementAt(1), "A");
                String msgtoBeDecoded = v.elementAt(2);
                SecretKey keyS = (SecretKey) keys.get("KAS");
                Vector<String> decode = decrypt(msgtoBeDecoded, keyS);

                assert check(decode.elementAt(0), nonceA);
                assert check(decode.elementAt(1), "B");
                SecretKey keyAB = receiveSecretKey(decode.elementAt(2));
                registerKey(keyAB, "K");
                String encryptedMsgToB = decode.elementAt(3);



                Vector vToB1 = new Vector();
                vToB1.add(encryptedMsgToB);
                net.send(this, "B", vToB1);
                break;
            case 1 :
                receivedNum++;
                // check first
                assert check(v.elementAt(0), "B");
                // check second
                assert check(v.elementAt(1), "A");
                String msgtoBeDecoded2 = v.elementAt(2);
                SecretKey keyAB2 = (SecretKey) keys.get("K");
                Vector<String> decode2 = decrypt(msgtoBeDecoded2,
keyAB2);
                String nonceB = decode2.elementAt(0);

                Vector vToB2Enc = new Vector();
                Vector vToB2 = new Vector();
                SecretKey succ = (SecretKey) keys.get("SUCC");
                Vector vToBNonce = new Vector();
                vToBNonce.add(nonceB);
                vToB2Enc.add(encrypt(vToBNonce, succ));
```

```
                vToB2.add(encrypt(vToB2Enc, keyAB2));
                net.send(this, "B", vToB2);

                String msg1 = generateMessage("FIRST BIT OF MESSAGE");
                String msg2 = generateMessage("SECOND BIT OF MESSAGE");
                Vector vToB3Enc = new Vector();
                Vector vToB3 = new Vector();
                vToB3Enc.add(msg1);
                vToB3Enc.add(msg2);
                vToB3.add(encrypt(vToB3Enc, keyAB2));
                net.send(this, "B", vToB3);
            }
        }

}
class B extends ComClass {
    Network net;
    String nonceBstore;

    public B (Network net){
        this.net = net;
    }



    public void processIncoming(Vector<String> v){
        switch (receivedNum) {
            case 0:
                receivedNum++;
                // check first
                assert check(v.elementAt(0), "A");
                // check second
                assert check(v.elementAt(1), "B");
                // check third
                // assign fourth
                SecretKey key = (SecretKey) keys.get("KBS");
                String msgToBeDecoded = v.elementAt(2);
                Vector<String> decode = decrypt(msgToBeDecoded, key);

                SecretKey keyAB = receiveSecretKey(decode.elementAt(0));
                String source = decode.elementAt(1);
                registerKey(keyAB, "KAB");

                Vector vToAEnc = new Vector();
                String nonceB = generateMessage("nonceB");
                nonceBstore = nonceB;
                vToAEnc.add(nonceB);
                Vector vToA = new Vector();
                vToA.add(encrypt(vToAEnc, keyAB));
                net.send(this, "A", vToA);

                break;
            case 1:
                receivedNum++;
```

```
                    // check first
                    assert check(v.elementAt(0), "A");
                    // check second
                    assert check(v.elementAt(1), "B");
                    String msgToBeDecoded2 = v.elementAt(2);
                    SecretKey key2 = (SecretKey)keys.get("KAB");

                    Vector<String> decode2 = decrypt(msgToBeDecoded2, key2);

                    SecretKey succ = (SecretKey) keys.get("SUCC");
                    Vector vNonce = new Vector();
                    nonceB = nonceBstore;
                    vNonce.add(nonceB);



                    assert check(decode2.elementAt(0), encrypt(vNonce, succ));
                    break;
                case 2:
                    receivedNum++;
                    // check first
                    assert check(v.elementAt(0), "A");
                    // check second
                    assert check(v.elementAt(1), "B");
                    String msgToBeDecoded3 = v.elementAt(2);
                    SecretKey key3 = (SecretKey)keys.get("KAB");

                    Vector<String> decode3 = decrypt(msgToBeDecoded3, key3);
                    String msg1 = decode3.elementAt(0);
                    String msg2 = decode3.elementAt(1);
                    theLogger.info(msg1);
                    theLogger.info(msg2);
            }

        }
    }
    class S extends ComClass {
        Network e;

        public S (Network e){
            this.e = e;
        }

        public void processIncoming(Vector<String> v){
            switch (receivedNum) {
                case 0:
                    receivedNum++;
                    // check first
                    assert check(v.elementAt(0), "A");
                    // check second
                    assert check(v.elementAt(1), "S");

                    assert check(v.elementAt(2), "A");
```

```
                String destination = v.elementAt(3);
                String nonce = v.elementAt(4);

                SecretKey keyA = (SecretKey) keys.get("KAS");
                SecretKey keyB = (SecretKey) keys.get("KBS");

                Vector vToAEncrypted = new Vector();
                vToAEncrypted.add(nonce);
                vToAEncrypted.add(destination);
                SecretKey key = generateSharedKey();
                registerKey(key, "K");
                vToAEncrypted.add(sendKey(key));
                Vector vToBEncrypted = new Vector();
                vToBEncrypted.add(sendKey(key));
                vToBEncrypted.add("A");
                vToAEncrypted.add(encrypt(vToBEncrypted, keyB));

                Vector vToA = new Vector();
                vToA.add(encrypt(vToAEncrypted, keyA));
                e.send(this, "A", vToA);

                break;
        }
    }
}
```

Elyjah output (NeedhamSchroeder.lysa)

```
(new KAS) (new KBS) (new SUCC) (
!(new nonceA) <A,S,A,B,nonceA>.(S,A;msgtoBeDecoded). decrypt
msgtoBeDecoded as {nonceA,B;K,encryptedMsgToB}:KAS in
<A,B,encryptedMsgToB>.(B,A;msgtoBeDecoded2). decrypt msgtoBeDecoded2 as
{;nonceB}:K in (new msg1) (new msg2) <A,B,{{nonceB} : SUCC} :
K>.<A,B,{msg1,msg2} : K>.0
|
!(A,B;msgToBeDecoded). decrypt msgToBeDecoded as {;KAB,source}:KBS in
(new nonceB) <B,A,{nonceB} : KAB>.(A,B;msgToBeDecoded2). decrypt
msgToBeDecoded2 as {{nonceB} : SUCC;}:KAB in (A,B;msgToBeDecoded3).
decrypt msgToBeDecoded3 as {;msg1,msg2}:KAB in 0
|
!(A,S,A;destination,nonce).(new K) <S,A,{nonce,destination,K,{K,A} :
KBS} : KAS>.0
)
```