SRI INTERNATIONAL

SRI Technical Report

Cryptographic Protocol Generation From CAPSL

Jonathan Millen and Frederic Muller
Computer Science Laboratory

SRI-CSL-01-07, December 2001

Supported by the National Science Foundation Award No. CCR-0082500

333 Ravenswood Avenue • Menlo Park, CA 94025-3493 • (650) 859-2000

Cryptographic Protocol Generation From CAPSL[1]

Jonathan Millen and Frederic Muller
Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA
{millen}@csl.sri.com

Abstract

Java code is generated to implement cryptographic network protocols specified abstractly in CAPSL. The translation source is the CAPSL intermediate language CIL, and the output includes code for each party and for a demonstration environment server that permits user examination and control of communication. Standard Java cryptographic providers are used.

[1]Supported by the National Science Foundation Award No. CCR-0082500

Contents

Chapter 1

Introduction

1.1 Background

One of the most theoretically challenging and practically important problems in modern computer science is the analysis of cryptographic authentication protocols. Connection establishment protocols on the Internet use cryptographic techniques for privacy, to authenticate the identities of communicating parties and their access rights, and to accomplish more complex ends such as auctions, elections, and fair exchanges in a protected way. Their security depends not only on the strength of the cryptographic algorithms, but also on the logical design of the message exchange. Protocols are designed to defeat attacks in which the intruder is assumed able to intercept and forge messages, but unable to decrypt information without the appropriate key.

Various formal methods exist for protocol security analysis, including inductive proofs, model checking, symbolic state search, and specialized logics. Although the protocol analysis problem is undecidable in an environment where the number of instances of the protocol is unbounded, the existing methods have been able to verify, or find flaws in, many particular protocols. These analysis methods are not applied to the executable code of a protocol; they are applied to a formal specification of it, in a language appropriate for the method or its associated software tools.

A general cryptographic protocol specification environment developed at SRI includes a high-level specification language, CAPSL, that can be translated into the input language for several (perhaps almost all) protocol analysis tools [DM00, DMR00]. In CAPSL, a protocol is specified in a textbook-like style as a sequence of messages, making use of abstract data types and encryption operators. The CAPSL environment includes an intermediate language, CIL, based on a multiset rewriting rule model originally suggested by researchers at Stanford. The CIL representation of a protocol is simple and general enough to serve as a common form. A protocol in the CIL form can be analyzed by any tool for which there is a translator (called a "connector") from CIL to the tool input language, as shown in Figure 1.1. Experimental connectors for several tools have been written.

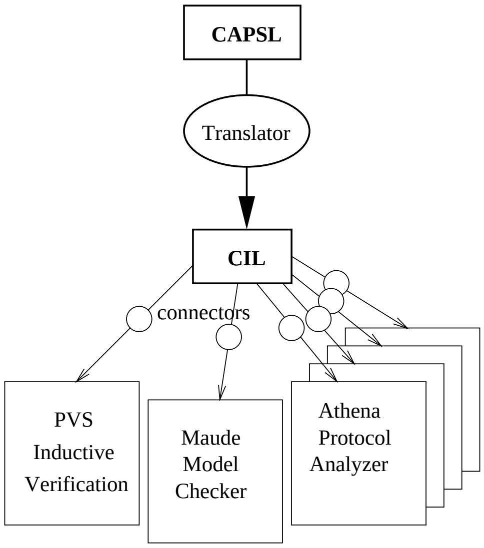

Figure 1.1: CAPSL and CIL Translation

The basic idea of this work is to write a code generator from CIL. The code generator is something like a connector, except that its target language, Java, is intended for execution rather than security analysis. This gives us a picture like Figure 1.2. For each protocol party, an object of a "CILProtocol" Java class is produced, and each of these is further translated into an executable Java class. The execution environment has further structure described later in this chapter.

Assuming that the code respects the semantics of the CIL, an analysis or proof of a protocol based on the CIL representation should apply to the generated code, since the analysis was performed on the same CIL representation. This is a different idea from "correctness by construction," since

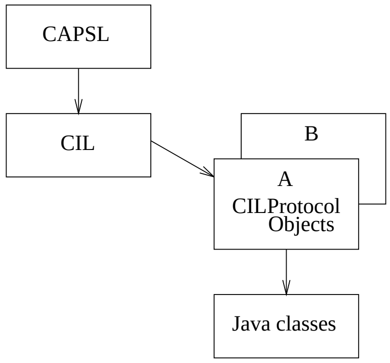

Figure 1.2: Code Generation from CAPSL

flawed protocols can be represented in CIL and have code generated for them. The approach pursued here might be described as "correctness by equivalence," since the code should be equivalent in formal structure to the CIL version, which can be translated and examined in some target analyzer.

In this project we concentrate on the first step for correctness by equivalence, and that is to establish the feasibility of translating the abstract CIL representation into running code. Our approach is to generate Java code by making maximal use of existing Java APIs (application program interfaces) for communication and encryption tasks.

1.2 CAPSL and CIL

1.2.1 CAPSL Overview

Here is a small example of a protocol specification in CAPSL, illustrating several of the language features. It looks like a strongly typed program, with variable declarations. Unlike an ordinary computer program, the body is a sequence of protocol messages, and it has a list of security goals.

```
PROTOCOL OneMessage;
VARIABLES
    A, B: PKUser;
    K: Skey, FRESH, CRYPTO;
ASSUMPTIONS
    HOLDS A: B;
MESSAGES
```

```
    A -> B: {A,K}pk(B);
GOALS
    SECRET K;
END;
```

This protocol has only one message, in which principal A sends a newly generated key K to another principal B. The key K is concatenated with A and encrypted using B's public key.

Protocol variables are typed. There is a type hierarchy. For example, the type `PKUser` given for variables A and B is a subtype of `Principal`. The public key function `pk` is defined on objects of type `PKUser`.

The most common types, and the operators that are defined on them, are specified in a file called the *prelude*. The CAPSL user may also supply additional abstract data type declarations in separate *typespec* modules. In this prototype code generator, we will handle most of the types in the prelude, but we do not yet handle user-defined types.

Protocol variable declarations may specify *properties*. For example, the property FRESH of K means that the value of the variable is a new value generated for each session, and has not (with cryptographically high probability) been used before by the principal generating it. The property CRYPTO means that the value of the variable is unguessable (because it comes from a large domain and is chosen in an unpredictable way).

The HOLDS assumption means that when an A process begins, it is initialized with the identity of a principal B to communicate with.

One security goal is specified: that the new key K generated by A is kept secret from any attacker, assuming that the attacker is not A or B.

1.2.2 The CAPSL Prelude

The prelude includes typespecs for basic and concatenated message fields, and generalized symmetric-key and public-key encryption. Fields may be concatenated with either `cat`, which is associative, or `con`, which is not.

Here is an example of some related typespecs found in the prelude, defining the basic public-key type Pkey, the RSA-like public-key encryption/decryption function `ped`, and the functions for retrieving the public and secret keys (both of Pkey type) from a principal of subtype PKUser.

```
TYPESPEC PKEY;
TYPES Pkey;
END;

TYPESPEC SPKE;
IMPORTS PKEY;
FUNCTIONS
    ped(Pkey, Field): Field;
    ped(Pkey, Atom): Atom;
END;

TYPESPEC PPK;
IMPORTS SPKE;
TYPES PKUser: Principal;
FUNCTIONS
    pk(PKUser): Pkey;
    sk(PKUser): Pkey, PRIVATE;
VARIABLES
    A: PKUser;
    X: Field;
AXIOMS
    ped(sk(A),ped(pk(A),X)) = X;
    ped(pk(A),ped(sk(A),X)) = X;
    INVERT ped(pk(A),X): X | sk(A);
    INVERT ped(sk(A),X): X | pk(A);
END;
```

The axioms are used in different ways by the CAPSL translator and by analysis tools. The code generator, like many analysis tools, has its own understanding of the cryptographic functions which is consistent with these axioms but does not use them formally.

The prelude also defines a symmetric key type Skey with the encryption function `se` and the decryption function `sd`, a hash `sha`. There is a subtype Node of Principal such that a pair of nodes have a shared symmetric key found by `msk`.

1.2.3 CIL Overview and Translation

The CAPSL specification is not immediately suitable for translation to code, because the message list does not explicitly indicate the separate actions of the processes in the A and B roles. The translator from CAPSL to CIL extracts individual state transitions for A and B and expresses them in a term-rewriting model. This form is useful both for abstract vulnerability analysis and for translation into executable code.

The meta-notation of Cervesato, et al. [CDL+99] was adapted for CIL. Their notation, according to the authors, could be regarded as either an extension of multiset rewriting with a kind of existential quantification, or a Horn fragment of linear logic. The logical style of this formalism made it suitable to serve as the language in which to express the semantics of CAPSL. Furthermore, the term-rewriting aspect corresponded well with the modeling approach taken by Denker, Meseguer, and Talcott with Maude [DMT98], as well as other analysis approaches. CIL may be regarded as a notational variant of the linear logic formalism in which certain specific conventions have been used to set up protocol models derived from CAPSL specifications.

The core of a CIL specification is a collection of rules specifying state transitions. A network is viewed as a multiset of "facts" representing either messages that have been sent, or the local states of processes executing roles in the protocol. The state transition rules indicate how processes are initiated and change state.

The abstract form of a CIL rule representing a transition in which a process in role B receives and acknowledges an encrypted nonce might be

$$B_0(B), M(A, B, \{N\}\mathtt{pk}(B))) \rightarrow B_1(B, A, N), M(B, A, \{N\}\mathtt{pk}(A))$$

and the corresponding translator output text in functional notation would be

```
rule(
  facts(state(roleB,0,terms(B)),
        msg(A,B,terms(ped(pk(B),N)))),
  ids( ),
  facts(state(roleB,1,terms(B,A,N)),
        msg(B,A,terms(ped(pk(A),N))))
)
```

The `state` fact distinguishes the role name `roleB` from the dummy variable B, and it indicates the current state of the process (which changed from 0 to 1). The state contains the local memory of the process as a list of terms, which can expand as the process reads information from received messages. One message is received and a reply is transmitted in this state transition. The empty `ids` list would include any new nonces or session keys generated for this transition.

Besides the transition rules, the output of the CAPSL-to-CIL translator includes a symbol table and a *slot* table. The symbol table declares types of protocol variables and the type signatures of operators. Examples of symbol table entries are:

```
symbol(ped,op,ids(PKUser),Pkey,props()),
symbol(A,pvar,ids(),PKUser,props()),...
```

The slot table maps argument positions in state memory lists to protocol variable names. This is necessary because variables appearing in rules are dummy variables that do not necessarily have the same names as protocol variables (although the translator chooses them that way). A typical slot table entry would be `slot(A,roleB,2)`, meaning that the state of the role B process stores the value of protocol variable A as the second term in the memory sequence.

The entire CIL output is a functional expression of the form

```
CILspec(
  symbols(
    symbol(...)...
  )
  axioms(...)
  slots(...)
  rules(
    rule(facts...)...
  )
  goals(...)
  envs(...)
)
```

Axioms and goals are not used by the code generator. The optional "envs" (environment) section declares constants used in model checking, and it may be used by the code generator to give names to the parties in the protocol in the demonstration environment.

1.3 Resources

1.3.1 The Berkeley Project

We got off to a running start by examining the Java translator developed under the Automatic Protocol Generation (APG) project at Berkeley, undertaken by Adrian Perrig and Dawn Song [PS00], with support by Doantam Phan. APG generates programs in its own CAPSL-like language and in the language of Athena, a protocol analysis tool that uses a related but more CIL-like representation based on the strand space model [THG99]. The idea of their project was to rapidly generate a very large number of randomly constructed programs aimed at satisfying a set of security goals, and filter them for correctness using Athena to find one or a few successful protocols. A subsequent phase of the project translated these protocols to executable Java code from Athena, using APG input to provide comments.

The Berkeley project extracted, from the Athena source file, information concerning each of the parties involved, and created intermediate objects representing the messages received and sent by each party. Later on, these intermediate objects could be interpreted to generate executable code. This code gave methods for each party to execute rounds. A round is an implemented state transition; it consists of receiving a message as a byte array (possibly empty), interpreting it, and returning a new byte array as a reply message (also possibly empty). Some test code was also provided to exercise the protocol by making every party use the appropriate round methods in the appropriate order.

This code was helpful as a starting point, but it was limited because it did not include state information in the intermediate object form, and it was not designed to be extensible with regard to cryptographic and other computational primitives. Our objective was to make a more general, extensible translator that took advantage of the expressiveness of CIL. This code also did not create any actual network communication.

A connector already existed to translate CIL files to the Athena language [Mil00]. So, as can be seen in Figure 1.3.1, there already existed a "path" going from CIL to executable Java code, with an intermediate step in Athena.

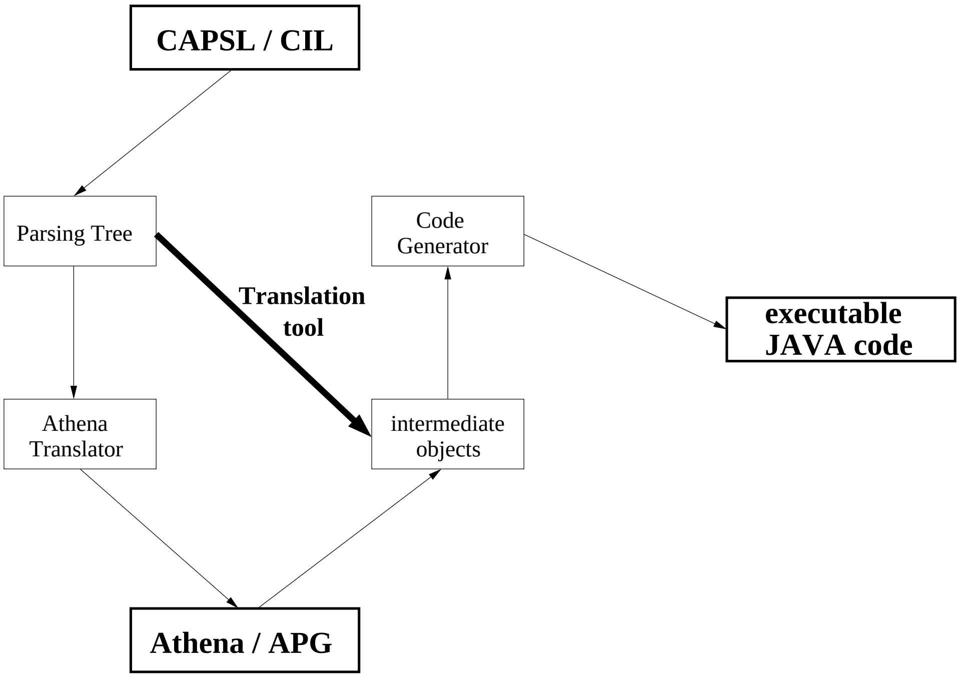

Figure 1.3: The Existing Framework

Then, the next idea was to skip the Athena stage. We could use the parse tree produced internally by the connector, and go directly to the Athena internal ACGProtocol object, still using the Berkeley code generator.

Later, as will be seen, several problems concerning the generality and usefulness of the code generator have made us change this structure profoundly. We designed our own internal CILProtocol object, which preserves state information, and reduced the role of the code generator to a simple rewriting, most of the cryptographic work being done elsewhere.

1.3.2 Cryptographic Providers in Java

In Java, there exists a set of abstract classes to represent cryptographic operations (e.g., Cipher, KeyGenerator, Mac). Providers are supposed to provide implementations for these classes in the security package. Sun supplies two main security providers : JCE and JSSE. Together, they provide a solid implementation of most of the common algorithms for various operations. We used Diffie-Hellman key agreement, DES encryption and key generation, the keyed hash function HmacSHA1, RSA key pair generation, and others.

The list of the provider classes currently used (which also defines the list of acceptable algorithms for every function) is maintained in the Security class. A new provider can be added by the method addProvider(). Here is an example of what to do in the case of JCE:

Provider sunJCE = new com.sun.crypto.provider.SunJCE();
Security.addProvider(sunJCE);

Once the provider has been added, it is possible to use new algorithms for many cryptographic objects. For instance, to create a new RSA key pair, once an adequate provider has been added, the code looks like:

KeyPairGenerator kpGen = KeyPairGenerator.getInstance("RSA");
KeyPair keyPair = kpGen.generateKeyPair();

The most commonly used security objects in our code are:

- KeyGenerator
  This class is used to generate a single secret key. Many algorithms are available but we decided to always use DES.

- KeyPairGenerator
  This class is used to generate a key pair. It is used for public/private key generation with the algorithm RSA, but also for key agreement, with the Diffie-Hellman algorithm.

- Cipher
  This class is used to encrypt and decrypt byte strings. In our code it is used only with DES. It could also be used with RSA if we had an adequate provider for that algorithm, but that is not the case, as explained below.

- KeyFactory
  This class is mostly an interface tool between Key objects and their parameters and specifications. This class is useful to extract some parameters and send them over the network, or to rebuild key from parameters received through the network.

- KeyAgreement
  This class provides methods for a Key Agreement protocol. Our code uses it with the Diffie-Hellman algorithm.

- Mac
  This class provides keyed hash functions. We use it with the algorithm HmacSHA1.

- MessageDigest
  This class is used for hash functions. We use it with the algorithm SHA.

Unfortunately, some operations are not furnished by Sun JCE, like Public-Key encryption (and especially, the RSA algorithm). Other providers have implemented RSA encryption, like Cryptix. However, in the case of Cryptix, their latest release is compatible only with the old JCE 1.1, which was not compatible with our code. Finally, the last possibility would be to implement our own security provider. This is feasible, but for the use of the main cryptographic operations (like encryption), it requires a certificate from Sun, and that entails a long administrative process.

Therefore, our project uses only the providers JCE and JSSE. This means there is no implementation of public-key encryption (although we are generating RSA key pairs). This operation is replaced by a dummy encryption. It is up to a user to purchase an RSA license or to implement his own Java Security Provider if he actually wants to do the encryption. But, for the purposes of our project, we felt it was a waste of time to spend weeks trying to get some RSA encryption working.

1.4 Structural Approach

1.4.1 Code Structure

The translator code comprises four main packages:

- The parse.tools directory contains several classes that are used to turn the CIL file into a tree. Most of them come from the CIL parser and have not been changed.

- The cil.tools directory contains several methods to turn the CIL tree to an intermediate CILProtocol object per party (TreeTranslation.java), and later to turn this object to executable Java code (CILGenerate.java).
- The code.tools directory contains the CILObject class, as described in a later section. This class consists of tools to represent the variables that the generated Java code will handle.
- The root directory contains the cil2JAVA class, that is, the one called by the user to turn a CIL file into the several Java files representing the protocol. This class just uses the other packages to work. The root directory also contains tools to support the execution of the program, which will be described in a later chapter.

The generated code contains one class per party extending Party.java, and a class that represents the Environment server, extending Server.java. This last file is the one that should be executed – it starts the network, instantiates each party in the protocol, handles all the message exchanges in the protocol and even allows the user to interact with each party. Finally, some additional tools are implemented: a Directory server, which stores the shared information, like public keys or cryptographic parameters; and Computation and Interpretation classes to handle the computations each party needs to do, as well as the analysis of incoming messages and the generation of byte arrays for outgoing messages. These classes make use of the security providers for cryptographic operations. The structure of a party is shown in Figure 1.4.

1.4.2 The Environment

We devoted considerable thought to the best way to represent a protocol. The Athena project for code generation created party-specific files, like we do. But only one test program was running, calling methods in the party-specific classes, displaying on the screen what was currently going on.

We decided to have different processes for protocol parties running at the same time, and communicating through sockets. Thus, the idea of a central server came to us. This so called "Environment server" would start one process for each party involved. It would just forward the messages from one party to another. This server could also allow the user to interact

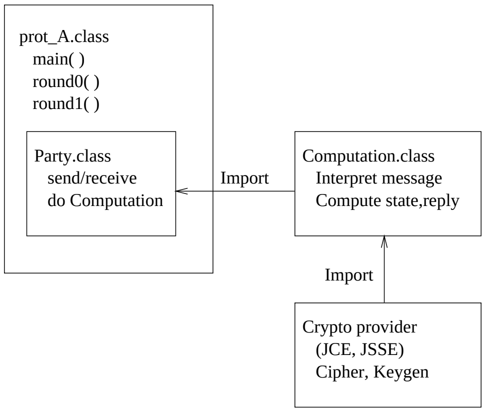

Figure 1.4: Party Structure

with the parties, and permit querying about the values of some variables, or even altering the content of the messages. This might even extend to allow the user to simulate a man-in-the-middle attack, by directly composing his own message. Although we did not have time to go very far in that direction, we took a first step by allowing users to query parties for variables, and composing new byte arrays as messages. This is not extremely useful to simulate attacks since nothing is done to support encryption or other computations, but this facility can be used to check that the protocols really work: if you alter one byte of a message, for example, it can appear a few rounds later that some party will notice that the value of some received nonce no longer corresponds to its expected value, and the protocol will abort.

The environment created by the code generator includes an Environment server, one process for each party, and a "Directory" server, as shown in Figure 1.5. The Directory server is essentially a database server for shared information such as encryption parameters that must be agreed upon by communicating parties prior to a protocol session. In the demonstration environment, shared data is generated in an initialization phase and collected by the Directory server.

The Directory server is intended to stand in place of a resource in a real implementation that stores long-term protocol-related information for each user. Such information could be stored locally, or in some cases it could be obtained through another network interface to a certificate authority or other separate network resource.

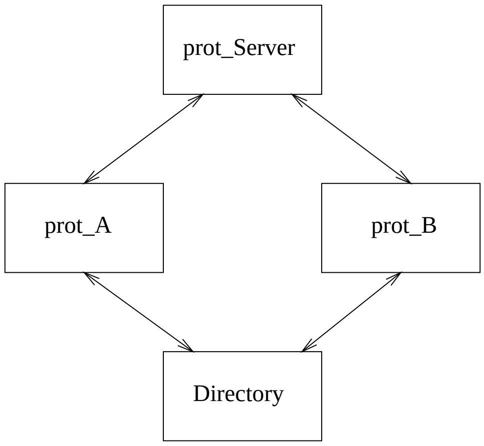

Figure 1.5: Demonstration Environment

Similarly, the Environment server is only for demonstration purposes, but it stands in place of external interfaces that call and communicate with the protocol processes, such as the user processes that initiate the protocol and the Internet interface for communication to other parties. Thus, the environment in Figure 1.5 would be replaced by the environment in Figure 1.6 for actual use, with no Environment server and implementation-specific changes confined to the Directory server.

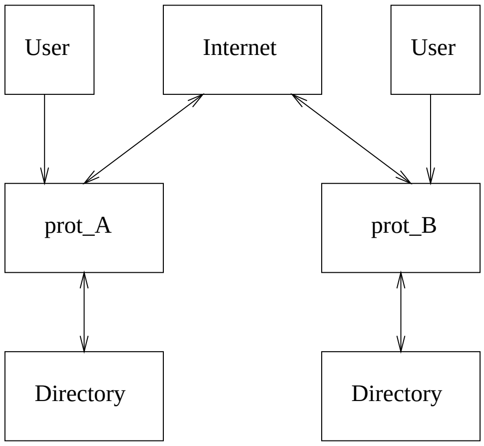

Figure 1.6: Application Environment

1.4.3 The User Interface

In this section are screenshots of an execution of the protocol NSPK.

The execution begins by starting the Environment server from a terminal. This server starts a Directory server (this role will be described more precisely in another chapter), and two processes, one for Alice, one for Bob. Then it waits until everyone is online to send Bob's identity to Alice. Finally, when everybody is ready, the protocol starts and the server switches to the mode "Router" where it only receives and forwards the messages of the protocol, with a possible user interaction in between.

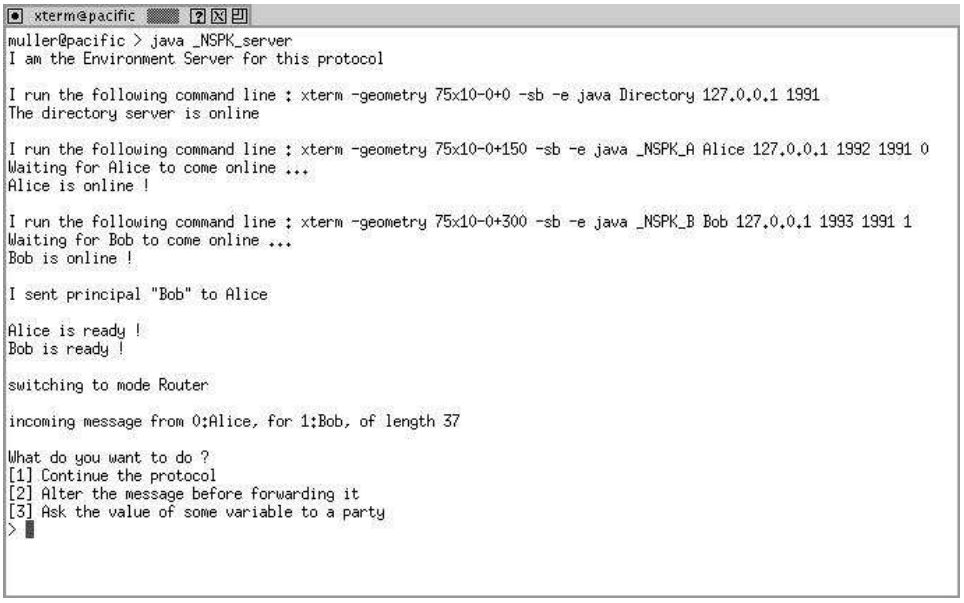

Figure 1.7: Environment Server

As soon as each party is started by the Environment server, it creates UDP sockets to communicate with the Environment and Directory servers. Then, it waits until it receives some variables it might need to receive before the execution of the protocol. Finally, it starts running the protocol NSPK.

The next three figures, 1.4.3 through 1.4.3, show screenshots of the Environment server, of one of the parties, and finally an overview of the protocol execution, which includes four windows: one for each of the two parties, one for the Environment server, and one for the Directory server.

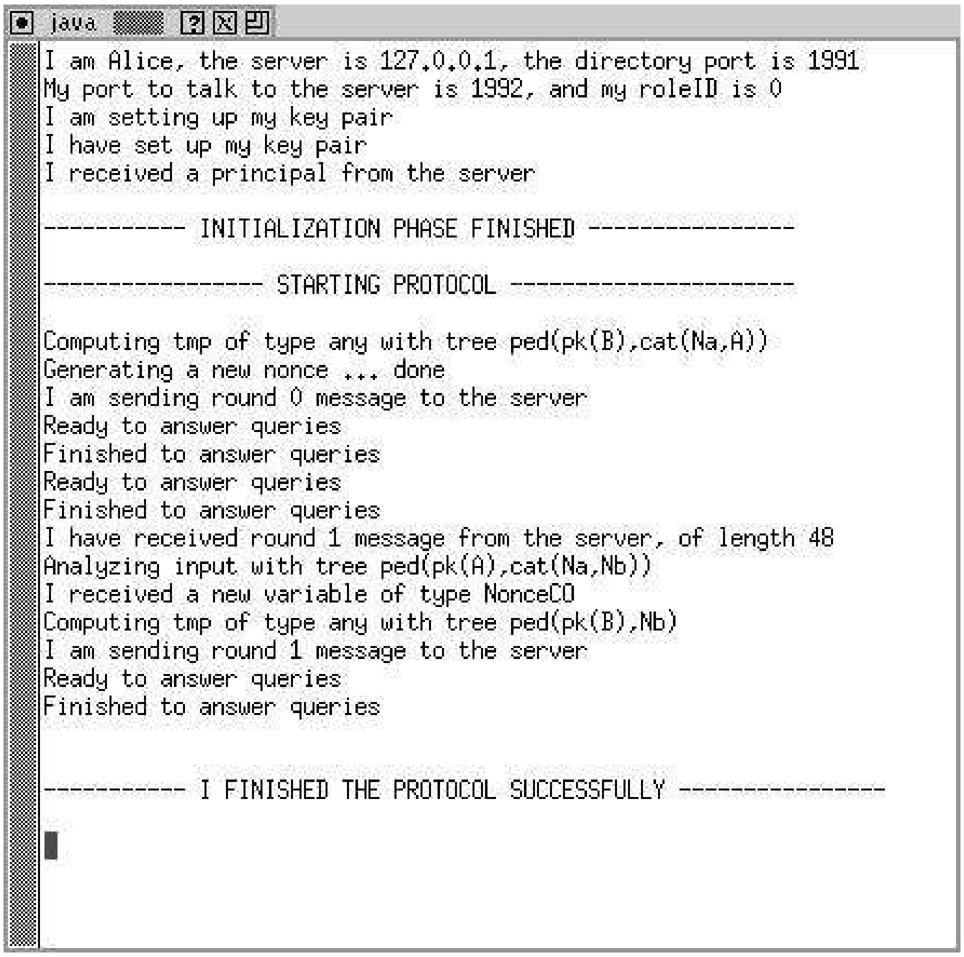

Figure 1.8: The window representing Alice

Figure 1.9: Overview of the protocol

Chapter 2

Translation Method

The code generation from CIL is split into two main steps: creating a CILProtocol object from the CIL file, and code generation.

Parsing the CIL into an internal parse tree was easy because a CIL already existed prior to this project, to support connector writing. The main class, treeParse, turns a CIL input file into a parse tree in the form of a LabelNode object. A LabelNode object is a labeled tree node; it has a string as a label and a possibly empty sequence of child nodes. The structure of the parse tree is exactly what one would expect from the functional expression form of the CIL: the function symbol for each term is a node label, and the arguments become the child nodes. Methods exist to relabel a node and to add, examine, and delete its children.

2.1 Creating a CILProtocol Object

Creating the CILProtocol object from the parse tree is done by the TreeTranslation class in the cil.tools package. A CILProtocol object has a vector of rounds for one party; each round contains the messages and state change information from one CIL rule.

The class TreeTranslation will first separate the information by parties involved in the protocol. To do that it first creates a new CILProtocol object for each of them, and initializes some basic information about the party (e.g., name, variable initially held, slots). The second step is to build the rounds for the current party.

2.1.1 Separating the rounds

A round is a state transition corresponding to a rule in the CIL file. Thus, a round corresponds to a distinct step in the execution of the protocol by the given party. It usually consists of an incoming message, an outgoing message, and some computation to do. We extract this information from the rules part of the CIL parse tree.

Assuming that the rules are always written in increasing order of state number for each party, we can read the rules table and build on the fly the rounds for each party. This ordering property is normally obeyed by the CIL translator. Then, the problem is to make the program able to see what new computations need to be done at that round. To do that, we first build a vector corresponding to the state of the current party after that round, by just reading the second (next) state subtree of the actual rule.

Anything here that is a function call needs computation. This computation must be done either at the beginning of the round if all parameters are already known, or between the phase of receiving a message and the phase of sending a message at that round.

2.1.2 Representation Types

The resulting CILProtocol object contains three very important vectors: inRound, outRound, and computeRound, containing all the information about the rounds for the current party, sorted by index in these vectors. A message (received or sent) is always represented by a PObject object.

The PObject class represents, in a uniform way, any kind of protocol message field that might occur as the value of a protocol variable. It is used also to represent messages. This class contains fields for name, type, parse tree, and a boolean flag to say if this object needs some computation to be built. For instance, when we send the message `cat(Na,Nb)`, we create a PObject with no name, type Field, and the binary tree labeled "cat" with two children *Na* and *Nb*, and the flag for computation *true*.

The result of this phase is to build a vector of CILProtocol objects, one for each party involved in the protocol. Each element of the array will later be used by CILGenerate to generate runnable Java code.

2.2 Code Generation

The code generation step consists in turning the CILProtocol object for each party into a CILGenerate object, which has methods to create the corresponding code, and a protocol-specific Party class. In addition, the cil2JAVA translator will have to create a protocol-specific Environment server class from information about the parties. This class contains the initialization of the parties (with their names, and their type), as well as the generation of variables some parties might need to hold initially prior to the protocol execution. Since the actual names of the parties need to be known (just to give them a name in the protocol), we try to find an environment statement that gives them a name. If it is not the case, the user is prompted to give a name to each party, as it can be seen in the example of code generation below.

So, for instance, for the protocol NSPK (Needham-Schroeder with Public Key), three Java files are generated: one to represent the Environment server (called "_NSPK_server.java"), and classes for party A and for party B (called respectively "_NSPK_A.java" and "_NSPK_B.java"). The first one initiates the protocol and starts one process for A and one process for B. It sends the name of principal B to principal A (since it appears in the holds statement). The last two files have just one method for each round, and execute them sequentially.

To generate such party files, most of the work is done by the method generateProtocol() of the CILGenerate class. This method first puts a comment at the beginning of each party file. Then, it has a 'main' method that puts all needed variables, with their type, into a Hashtable object. The created class extends the class Party (which will be described more precisely in the next chapter). Therefore, the generated code for the rounds consists only of calling other methods in the Party class to compose a message, receive a message, set up variables, and so on.

Consequently, the code generation is almost independent of the type of objects handled. For instance, it does not complain if the CIL file tries to encrypt a message with a boolean. Only during execution can such mistakes can be noticed. This is covered in the next chapter.

The advantage of that approach is that the code generation does not need to be changed every time one wants to add new functions or types. All that will need to be changed is possibly the CILObject class for possible types, and the Computation class for the possible functions.

We give one example of generated code for a very simple protocol before continuing.

2.3 Sample of Generated Code

Let us consider the protocol NRF for authentication using nonreversible functions a,s it appears in the Clark-Jacob library [CJ97], in Section 6.1.5.

```
PROTOCOL NRF;
VARIABLES
        A, B: Node;
        Ra, Rb: Nonce;
        K: Skey, CRYPTO, FRESH;
        Kab: Skey;
DENOTES
        Kab = msk(A,B): A;
        Kab = msk(B,A): B;
ASSUMPTIONS
        HOLDS B: A;
MESSAGES
        1. B -> A: B, Rb;
        2. A -> B: A, {sha(Rb), Ra, A, K}Kab;
        3. B -> A: B, {sha(Ra)}K;
GOALS
        SECRET K;
        PRECEDES A: B | Rb, Ra, K;
        PRECEDES B: A | Rb, Ra, K;
END;
```

Two parties A and B involved; therefore two party-specific files are generated. The file for A is:

```
import cil.tools.*;
```

```
import code.tools.*;
import java.security.*;
import javax.crypto.*;
import java.util.Vector;

public class _NRF_A extends Party {
   static private int role;
   static private KeyCO tmpKey;
   static void main(String[] args) throws ProtocolException {
      try {
         initializeNetwork(args);
         initializeDataStructure("msk");
         allVars.put("K", "KeyCO");
         allVars.put("B", "PrincipalCO");
         allVars.put("A", "PrincipalCO");
         allVars.put("Kab", "KeyCO");
         allVars.put("Rb", "NonceCO");
         allVars.put("Ra", "NonceCO");
         assignedVars.put("A", new PrincipalCO(Party.myName));
         ready();
         byte[] buf;
         buf = receiveBytes("round 0 message");
         buf = checkHeader(buf);
         buf = round0(buf);
         role = queryRoleID(((PrincipalCO)assignedVars.get("B"))
                                             .getPrincipal());
         buf = addHeader(role, buf);
         sendMessage(buf, "round 0 message");
         buf = receiveBytes("round 1 message");
         buf = checkHeader(buf);
         round1(buf);
         closeNetwork();
      } catch (ProtocolException e) {}
      for (;;) {}
   }

   public static byte[] round0(byte[] input)
        throws ProtocolException {
      processIn(input, "cat(B,Rb)");
      doComputation("Kab", "msk(A,B)");
```

```
      return processOut("cat(A,se(Kab,cat(sha(Rb),Ra,A,K)))");
   }

   public static void round1(byte[] input)
        throws ProtocolException {
      processIn(input, "cat(B,se(K,sha(Ra)))");
   }

}
```

We can observe the way incoming messages are analyzed (by calling the method processIn() in the superclass) and the way outgoing messages are created (with the method processOut()). For instance, here, the computation of Kab in round 0 comes only after the reception of the first message by A, because A needs to know first the identity of B.

There is a similar file for B. Finally, the file that is generated for the Environment server must initialize two parties, called Alice and Bob, and to send to B the identity of Party A (which actually corresponds to Alice in that case). Therefore, the generated code looks like:

```
import java.security.*;
import javax.crypto.*;
import java.util.Random;
import code.tools.*;

public class _NRF_server extends Server {
   private static int numOfParty;
   private static Random randGen = new Random();
   private static String[] parties;
   private static String[] files;
   private static int port_S = 1991;

   public static void main(String[] args) throws ProtocolException {
      // protocol specific initialization
      numOfParty = 2;
      parties = new String[numOfParty];
      files = new String[numOfParty];
      parties[0] = "Alice";
      files[0] = "_NRF_A";
```

```
      parties[1] = "Bob";
      files[1] = "_NRF_B";
      initializeServer(parties, files, port_S);

      //protocol specific exchanges of variables
      sendPrincipal(0, 1);

      waitAndForward();

   }

}
_NRF_server.java (END)
```

In this example, we can see that the generated code is very short. Most of the real computational and tree analysis work is in fact done by the classes Party, Computation and Interpretation, which are independent of the protocol.

Now, we discuss how the execution of this generated code works, since most of the cryptographic work is done at that time.

Chapter 3

Code Execution

3.1 The CILObject Class

In the Berkeley project by Perrig, Phan and Song, there was a class called ACGObject, which was supposed to represent any kind of object that might be included in messages, which included keys, nonces, principals, and encrypted fields. Each type of object had a corresponding subclass, with standard methods to create the object from a byte array and output the object to a byte array. There was no subclass for Concatenated Objects, since they were directly handled by the generated code.

Because of concerns about generality, we decided to reduce the role of the code translation to a simple analysis of the type of the variables, while all the computation work would be done by another class called Computation. Therefore, we needed to have a class that could handle every kind of protocol variable, as they may appear in the CIL slot and symbol tables. Therefore we replaced this ACGObject class by our own CILObject class, with subclasses for keys, nonces, principals, fields and booleans, as seen in Figure 3.1 below. To simplify the analysis, we grouped the possible types in the CIL file into these five categories. For instance, if a variable of type Pkey or Skey appears in the CIL, the code generator creates a variable of type KeyCO (Key CIL Object). Similarly, Nonce and Timestamp are regrouped in the NonceCO class. In the class CILUtils, several methods translate types as they appear in the CIL input into the implemented types, and *vice versa*.

All the possible types appearing in the standard CAPSL prelude have been

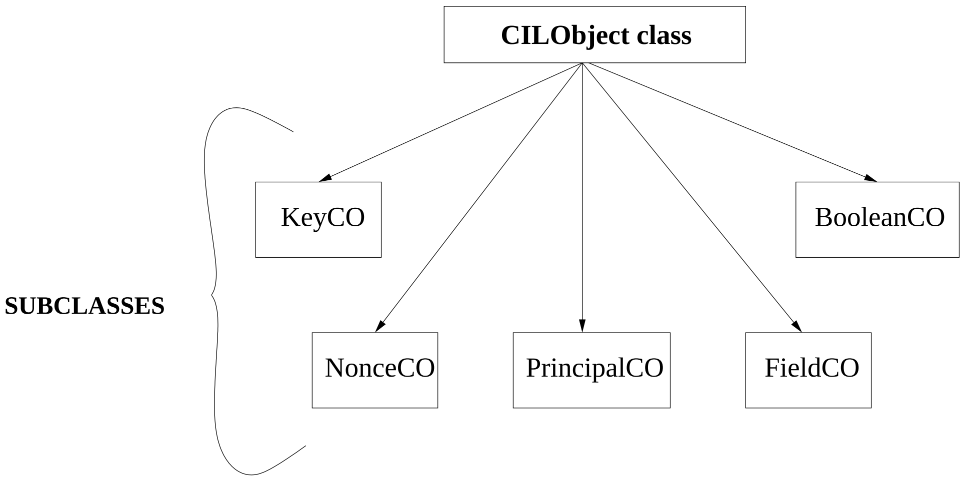

Figure 3.1: The CILObject Class

implemented, but if some other new type were to be created, there would be no problem for the code generator's point of view if it can be mapped to an existing type of CILObject. It is even possible to implement a new type. One must create a new subclass of CILObject, which must have the same CILObject methods (including outputBytes and inputBytes). In addition, the CILUtils class must be updated to associate CIL types with implementation types.

3.2 The Environment Server

The protocol-specific Environment server file specifies an extension of the class Server. It is protocol specific because it depends, for example, on the number of parties in the protocol. The Server class contains several standard methods:

- initializeServer()
  This method initializes the network, creating an array of sockets to communicate with each party. It also starts one process for each party as well as one process for the Directory server (which role will be explained in the next section). Each of these processes is run in a separate xterm window. The server waits until all the processes have

connected to proceed.

- Several methods to generate and send new variables. This is usually only used when there is a HOLDS statement in the CAPSL file.

- waitAndForward()
  This method loops forever. It is run as soon as the protocol initialization is done. When it switches to "router" mode, all the Environment server does is wait for incoming messages, forward them according to their headers, and query the user for an operation to do in between.

- Several functions for the possible interactions of the user. At the moment, only message modifications and queries to users have been implemented, but more things might possibly be done in the future to support simulation of message modification attacks.

3.3 The Directory Server

At some point, we had to face the problem of how to set up information such as encryption parameters that were assumed in the protocol to be shared initially by the communicating parties. For instance, in the case of public keys, we had to find a way for every party to access easily the public keys of other parties. Some protocols specify an exchange of certificates for this purpose, but others are written under the assumption that some public keys are already cached.

In our demonstration environment, it was necessary to generate shared public keys and other such parameters during the initialization phase. The next problem was how to generate code for parties that would work in the same way in both the demonstration enviroment and an application environment. We decided to have a Directory server running all the time, waiting for queries from any party, registering the information that parties want to share, and sending this information to another party when this is requested.

Since the Directory server receives its information from other parties during initialization, the server itself is independent of the protocol. It is started (in a new xterm window) every time we run a protocol. All it contains is a main() method that creates a ServerSocket object, waits for queries, and handles them. The information is stored in various arrays, in which the index corresponds to the party to which this piece of information belongs.

Therefore, there is a unique matching between index values, called Role Identification Numbers, and party names. We also decided to use these Role Identification Numbers as part of the header of our messages to specify the message destinations. Hence, every time a party comes online, it is supposed to register itself with the Directory server, by giving its name and ID, which were given to it as part of the command line by the Environment server. In an actual protocol implementation, these numbers would be replaced by IP addresses.

In an actual protocol implementation, the Directory server would be replaced by a local class that looks up shared data in a pre-configured table. In some cases, it could also invoke a subprotocol to obtain certificates from a fixed distribution point.

3.4 Representation of Parties

Each party involved in a protocol is represented by a specific class file. This file includes a main() that is invoked by the Environment server to start the protocol party process, and methods for each of the rounds. Each of these takes byte array as input (the incoming message) and also outputs a byte array (the outgoing message). As stated earlier, this class extends the class Party, and contains the following standard methods:

- initializeNetwork()
  This method initializes the party by creating sockets and registering it with the Directory server.

- Several network tool functions to connect and disconnect to the directory, and to synchronize the execution of the protocol with the Environment server, which is an important concern.

- initializeDataStructure()
  This method is used with every function appearing in the CIL symbol table input that has an argument of the type given to the current party (some subtype of Principal). For example, in the case of a PKUser, this method is called for functions "pk" and "sk", since each of these is a function mapping a PKUser to a key. Then, this function calls other functions to set up the data necessary to support it in the implementation. For instance, in the case of a Client, a secret key is be

generated. This information is systematically saved by the directory, but it is saved locally by the party only if the variable is present in the slot table for that party.

- Several functions to send, query, and receive variables of various types. These methods are used in the demonstration to exchange variables with the Environment server and the Directory server.
- processIn()
  This method is used to interpret an incoming message, by verifying its validity and/or setting up the value of some variables. This done by using the Interpretation class, which is described below.
- processOut()
  This method is used to compute an outgoing message, by using the Computation class, which is also described below.
- doComputation()
  This method is similar to processOut, since it also does some computation. The difference is that the output is not a byte array (for a message), but instead, its goal is to compute the value of some variables from other variables. Therefore, all this function does is change the Hashtable object corresponding to assigned variables.

3.5 The Computation Class

This class is responsible for doing any kind of computation a party might need to do. This includes computing some hash, generating a nonce, or querying the Directory server to get a public key. In that sense, this class is the only one that is really dependent on what kind of data our implementation accepts. This class contains a method doComputation() that takes a term parse tree as parameter. The method matches the root label and calls some other method, as a function of the named operation. For instance, it calls computeCat() if it sees the label 'cat'.

All these label-specific methods have the same structure. First, they recursively call doComputation() on the children of the root, put the results in a temporary table, retrieve these results, and compute the expected result with that. This mode of operation is represented in Figure 3.5.

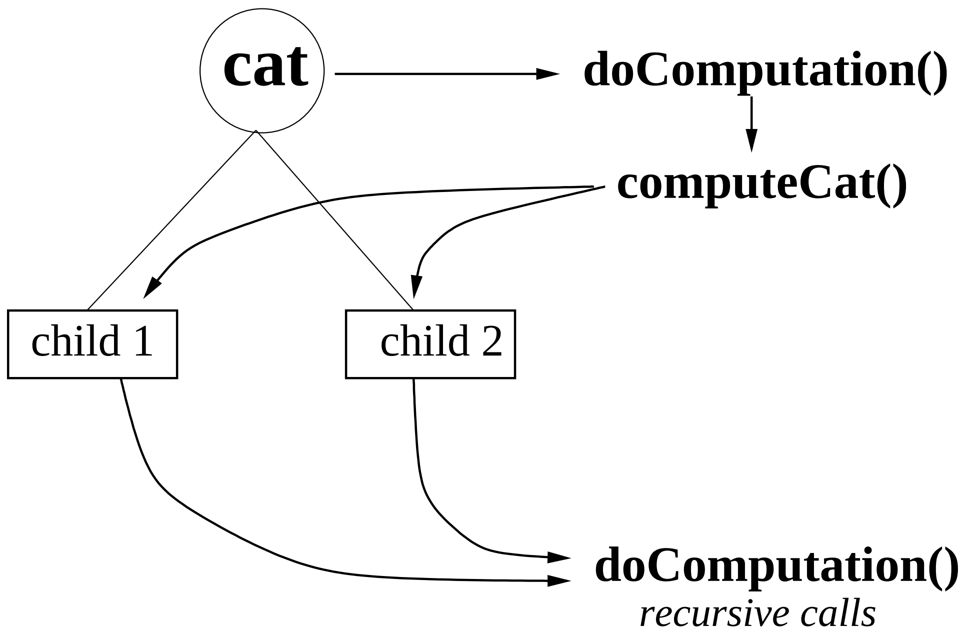

Figure 3.2: Example of Computation Using Recursion

Of course, some functions are not recognized in the present version. We did not have time to implement every possible function. However, we tried to support most of the CAPSL prelude. Thus, everything in the prelude but integer operations (e.g., addition, multiplication) are implemented. To do that, we had to make some assumptions concerning the implementation:

- Every key is of type KeyCO. However, we can check its actual type, by checking what class the field key (of type java.security.Key) in this class instantiates (for instance a public key is an instance of PublicKey).
- Every secret key is a 56-bit DES key. Therefore, all the functions that are supposed to output secret keys (e.g., sha, mac or kas) output DES keys. If this algorithm has to be replaced by something else, several things must be changed in order to preserve compatibility of functions with each other, as described in the Appendix.
- Every Public/Private key pair is an RSA key pair. Although the encryption is only a dummy encryption, due to license restrictions, we

believe that RSA is the most widely used assymetric encryption algorithm, so it is better to assume that we need RSA key pairs.

- KeyAgreement is implemented with the Diffie-Hellman algorithm. In CAPSL, key agreement is modeled with two abstract operators, 'kap' to produce the public key for a given secret key, and 'kas' to combine a public key with another secret key to produce a shared key. This means that when we call 'kap' with a secret key as a parameter, we usually want to create a Diffie-Hellman key pair (x, g^x), which is very difficult to do in Java from a practical point of view, if x is a DES secret key, as is usually the case with our code. Therefore, we had to cheat a little bit; the argument is not really used; we use x to generate a new key pair, (y, g^y), and we overwrite x with y. This is fine if x was newly generated in the protocol for the sole purpose of creating a Diffie-Hellman key pair. However, our code would not work if the new private key were to be used as a secret key for encryption before the Diffie-Hellman key pair was generated.

  Diffie-Hellman key exchanges also require other shared parameters – an exponent and a modulus. In order to force parties to use the same parameters, every time a public or a private Diffie-Hellman key is received or created, the party must query the Directory server to get these parameters, and only then can it build the key with them. Therefore, the Directory server generates the exponent and the modulus the first time it needs to answer such a query, and uses them for the rest of the execution.

- Some functions, like 'cat', have no particular requirement on the type of the arguments. For that kind of situation, we added a general field type, which is parsed as the string 'any', and which basically means that we do not really care about the type of the result; all we want is a byte array.

- Booleans are not message fields in CAPSL. They can be used only for computations that occur between the reception and the composition of messages.

3.6 The Interpretation class

The interpretation of incoming messages is a much more difficult problem than the composition of messages. As a matter of fact, an incoming message might play different roles – it could be sent to verify the values of some variables, but it could also be sent to give the values of some variables, or it could mix both. For instance, a message may contain a nonce as well as an encryption of the same nonce to authenticate the key, as in this example:

```
A -> B: Na,{A,Na}sk(A);
```

When this message is received by B, party B should already know the value of A and its public key. It learns the value of Na and uses the public key of A to validate the encrypted field, both for the identity of A and to match the encrypted and unencrypted values of Na in the message. All of this is implied by CAPSL, and the CAPSL translator uses a "receivability" algorithm to check that the implied computations are implementable, as it produces the CIL translation of the message.

Therefore, we need to understand what computations are implied when a message is received, as explained in the next section.

3.6.1 Computability and Receivability

The translator from CAPSL to CIL includes algorithms for "computability" and "receivability" to determine what computations are implied when messages are received (and sent). The algorithm used by the Interpretation class is based on these algorithms, although it has been modified to recognize the limitations of using particular Java cryptographic providers. It retains the computability and receivability portions, but handles invertibility with special cases.

We begin with some necessary terminology to describe our algorithms abstractly. In general, a boldface symbol is a sequence, so $\mathbf{x} = x_1, ..., x_n$ for some n. In some contexts we also use $\mathbf{x}$ to refer to the set of its components. $R(p)$ denotes the symbol of type Role corresponding to a symbol p of type Principal.

Accessibility. A function $f(\mathbf{y})$ is *p-accessible* if f is computable by the party that needs it. For example, B cannot compute the private key $\mathtt{sk}(A)$

of a different party. In CAPSL, accessibility is restricted using a property PRIVATE in the function declaration. In the code generator, accessibility is known for particular methods of a prespecified set of cryptographic providers.

Computability. In defining computability of a term, we assume that some terms are held—this is the set G—and we derive the set of additional variables X that are needed to compute the term. This gives a relative notion of computability. The principal p is mentioned below only because of the need to test accessibility.

t is *p-computable given G with H* if

1. $t \in G$ and $H = \emptyset$ or
2. t is a protocol variable and $t \notin G$ and $H = \{t\}$ or
3. $t = f(\mathbf{y})$, $f(\mathbf{y})$ is p-accessible, each y_i is p-computable given G with H_i, and $H = \bigcup_i H_i$.

We say that t is p-computable given G if t is p-computable given G with $\emptyset$. If Z is a set of terms, we say that Z is p-computable given G with $\bigcup_{t \in Z} A_t$ if each $t \in Z$ is p-computable given G with A_t.

As an example, consider $t = \texttt{ped}(\texttt{SK}(A), N)$. t is A-computable given $\{A\}$ with $\{N\}$, or given $\{A, N\}$ with $\emptyset$.

Invertibility. Invertibility is the property of certain functions that permits recovery of some of their arguments from knowledge of their values and perhaps other terms. Encryption functions are invertible and hash functions are not.

t is *p-invertible at i given G* if $t = f(\mathbf{y})$ and y_i is computable from t and the terms in G. For example, `ped(pk(A),X)` is A-invertible at 2 given $\{A\}$.

Invertibility is expressed in CAPSL using an INVERT axiom, but for code generation we currently test for individual known operators rather than interpret the axioms.

Receivability. If a term t is a variable or constant (a function with no arguments), receiving it means to compare it with the terms in the held set G and add it to G if it is not there. If t is compound, it must be either computable or invertible, and in the latter case the components extracted from it are received recursively. This process enlarges G to H.

t is *p-receivable given G to H* if

1. t is p-computable given G and $H = G$, or
2. t is p-computable given G with $\{t\}$ and $H = G \cup \{t\}$, or
3. We have:
   (a) $t = f(\mathbf{y})$ is p-invertible at some j given G and
   (b) $\mathbf{y}'$ is sequentially p-receivable given G to H', where $\mathbf{y}'$ is the maximum subsequence $y_{i_1}, ..., y_{i_k}$ such that t is p-invertible at i_j given G, and
   (c) if t is p-computable given H' then $H = H'$ else $H = H' \cup \{t\}$.

Sequential receivability expresses the notion that variables learned while receiving a message can be used to compute terms received later in the same message.

$\mathbf{y} = y_1, ..., y_n$ is *sequentially p-receivable given G to H* if, for $j = 1, ..., n$, y_j is p-receivable given G_j to H_j, where $G_j = H_{j-1}$ and $H_0 = G$ and $H_n = H$.

The success or failure of the sequential receivability test depends on the order of the sequence of terms, since the held set G is augmented as part of the process. A more forgiving definition would be able to rearrange the order to find one that works, and it could be implemented by making several passes over the sequence.

As an example, consider $t := \texttt{ped}(\texttt{sk}(A), N)$. t is B-receivable given $\{A\}$ to $\{A, N, t\}$. Upon receiving t the agent in role B not only learns the nonce N but also the entire term t, which might have to be saved for later use. The term t cannot be reconstructed from N and A because B does not possess A's secret key.

3.6.2 Implementation

Concretely, to test receivability of a message, the method doInterpretation() inside the class Interpretation performs the analysis described in the previous section: first, the actual label (function name) is noted and the parse tree is handled differently depending on it. In fact, we know some functions are not invertible, so they are receivable only if they are computable. On the other hand, we might need to test both computability and inversion, for instance in the case of an encryption.

To do that, we decided to let the Implementation class be a subclass of Computation. Hence, inside an interpretation, we can try to compute a subtree, catch eventual exceptions, and invert the function in case the computation failed. This is done, for instance, in methods interpretCat() and interpretPed().

If a party receives a new variable in a message, this variable is put into the hashtable of assigned variables. The values received are never directly printed on the output of the terminal, because this is not what a party does in the application environment. However, it is still possible to check the values of variables by querying a party directly, using the Environment server. If some received message does not match the expected computed value, an exception is raised.

Chapter 4

Conclusion

4.1 Summary

This project allows automatic generation of runnable Java code from the specification of a protocol written in CAPSL or CIL. The code runs in a demonstration environment, which includes a window showing activity of each party, and one Environment window where the user interacts with the protocol run. The parties communicate through the network, sending variables, messages or answers to user queries.

Our code includes two distinct phases: one to generate the Java code, which is independent on the implementation of functions because it just analyzes the CIL rules to see what operations need to be done by each party and at what time. Once the code is generated, the second phase consists in running it with the help of several classes that will specify how to handle types and functions that can appear.

We made efforts to implement as many things as possible. Most of the functions of the standard prelude for CAPSL have been included, except the operations on integers. The code generator seems to work well on most of the Clark-Jacob library.

4.2 Limitations

From a protocol prototyping point of view, one limitation is that the classes for individual parties have special features for the demonstration environment which would have to be removed or modified for actual use in an application environment. Also, this version of the generated code is inefficient because it runs by interpreting an abstract data structure.

Another concern is the lack of a public-key encryption implementation. The most commonly used such encryption algorithm is RSA. Unfortunately, it is not implemented by the Java providers JCE and JSSE (although they implement RSA key pair generation). We tried other ways to do that, but, for various reasons, they were not practical in the short term. So, the use of public-key encryption in our project is limited to a dummy encryption right now. It is possible to rewrite that part of the code easily, if one obtains an RSA license or has implemented a Java Security Provider that implements the Cipher object with the RSA algorithm.

A further limitation is the dependence of key objects on the algorithm used. We use DES for symmetric key encryption and Diffie-Hellman for key agreement. As discussed in the Appendix, changing that requires a deep look into the code. Finally, there was an assumption on Key Agreement that private keys used in this algorithm are not to be used for anything other than forming a shared key in the protocol.

Some assumptions were made on the CIL input at various steps in the project. There is an assumption on the order of fields in messages to guarantee receivability. We also assumed that rules and slots appear in the CIL file in a "natural" order, which is not required by the CIL definition but which is observed by the translator from CAPSL to CIL.

4.3 Extensions

Possible extensions of this project include the implementation of additional functions to make it more complete from that point of view. This might include implementation of arithmetic operations on keys (e.g., exponentiation, multiplication) to support protocols defined with unusual combinations of these operations that do not correspond to higher-level abstract operators in the prelude.

Another possible development would be to offer choices for encryption and related algorithms. Complete flexibility is not possible, since a Java provider is needed and must be known ahead of time to support the appropriate interface. Also, there are relationships between algorithms for various purposes. For example, a hash function is usually chosen in coordination with an encryption algorithm, since the hash is often used to produce a key. On the other hand, certain protocols use different algorithms and providers for different phases of the protocol.

It would be useful to have an easy way to generate an application environment version of a protocol, as an option or transformation on the demonstration version. This would require a different – but actually simpler – design for the Directory server. Another consideration for usable versions of generated protocols is efficiency, which would be accomplished by replacing interpretive activities by in-line code.

For purposes of security demonstrations, an interesting extension would be to simulate attacks from the Environment server window by allowing the user to intercept messages, get and temporarily save some variables, and synthesize messages. The user could compose messages in a CAPSL-like style, and they would be parsed and turned into actual byte-array messages. The ability of an attacker to decrypt and decompose messages would have to be supported.

Bibliography

[CDL+99] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *12th IEEE Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society, 1999.

[CJ97] J. Clark and J. Jacob. A survey of authentication protocol literature. `http://www.cs.york.ac.uk/~jac/papers/drareviewps.ps`, 1997.

[DM00] G. Denker and J. Millen. CAPSL integrated protocol environment. In *DARPA Information Survivability Conference (DISCEX 2000)*, pages 207–221. IEEE Computer Society, 2000.

[DMR00] G. Denker, J. Millen, and H. Ruess. The CAPSL integrated protocol environment. Technical Report SRI-CSL-2000-02, SRI International, 2000.

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *Formal Methods and Security Protocols*, 1998. LICS '98 Workshop.

[Mil00] J. Millen. A CAPSL connector to Athena. In H. Veith, N. Heintze, and E. Clarke, editors, *Workshop of Formal Methods and Computer Security*. CAV, 2000.

[PS00] A. Perrig and D. Song. A first step toward the automatic generation of security protocols. In *Network and Distributed System Security Symposium*. Internet Society, February 2000.

[THG99] J. Thayer, J. Herzog, and J. Guttman. Strand spaces: proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.

Appendix A

How to Add a New Function

The automatic generation of protocol-specific Java code works only as long as we have implemented real computation methods corresponding to all the concrete operations that must be performed during that protocol.

An important concern for the evolution of this project is the possibility to add new features easily, or to change them. Therefore, we will try to explain precisely in this appendix how to add (or modify) a function. Two main things need to be done: to adapt the possible variables, and to update the Computation class.

A.1 The Types of Variables

This project presently provides five types of variables: PrincipalCO, KeyCO, NonceCO, BooleanCO, and FieldCO, which represent all the possible kinds of variables that need to be handled. All of these classes extend CILObject as explained in a previous chapter.

Anyone who wishes to add some new function should first think about its argument and value types. If they can map to one of the five standard types above, nothing special needs to be done here. However, if some new type must be added, the following should be done:

- Go to the directory code.tools and create a new Java file for the new type.

- Make this new class an extension of the class CILObject.
- Implement the four standard methods that must be present in that class: getTypeID() that will return a byte representing the type of the object (see more details below), outputBytes() and inputBytes() to translate this object into a byte array, or the opposite, and equals() to test the equality of this object with another CILObject.
- Go to the cil.tools.CILUtils class. In that class, connections are made between strings read in the CIL input and corresponding implementation types.
- There, it is important to change the functions typeName(), typeCIL(), and bytesToObjects() as well as the array typname[]. These are all various tools to handle types in the code. What must be done exactly should be clear from the comments and from what is already implemented.

The next concern is the computation (and interpretation) of the new functions.

A.2 The New Functions

To add a new function 'F', two new methods should be implemented:

- computeF() that should take a tree (of type LabelNode), and two strings (one for the expected name, one for the expected type) as input. First, it should check that the expected type corresponds to the type that F usually outputs. Then, it should compute the values of the children of the LabelNode, and apply adequate operations to compute the value of F from its parameters. Finally, an entry should be added to the Hashtable tempVars, with the expected variable name as its key, and with the CILObject that was just computed as its value.
- interpretF() should take a tree and a byte array 'input' as arguments. It should also check first if the arguments match what is expected. Then, it should try to call computeF(). If an exception is raised, it should catch it, and do appropriate operations to invert F, eventually set up some values, and so on. Otherwise, it should verify that the

computed value matchs the argument input and raise an exception if that is not the case.

- Finally, one line should be added in Computation.doComputation() and Interpretation.doInterpretation() to redirect these functions to computeF() and interpretF(), respectively, when the term label is 'F'.

It is not really possible to say more about this step. The computation and interpretation are very dependent on the kinds of operations implemented. It may require some cryptographic tools, in which case reading some documentation on java.sun.com and using JCE and JSSE would be helpful. It may also require some byte or number handling, in which case, the standard libraries should be sufficient.

A.3 Extensions

This method to add new functions can be extended to the case in which a user would like to use different algorithms than what we used. For example, it is possible to replace DES, by rewriting the methods corresponding to the secret key handling (ped, sed, csk, ssk, msk, sha, mac). Most of the time, only the name of the algorithm would need to be changed. But the encryption could be made more realistic by adding macs, random bytes, or other extensions.

The situation is similar for Key Agreement (where we use Diffie-Hellman) and public-key operations (where we use RSA, but as explained earlier, we do not have an implementation, so we do not really encrypt messages). In all of these cases, the Directory server will also have to work differently, since it will no longer store the same kind of data if the algorithms change. For instance, right now, it has an array to store RSA public exponents, and one array to store an RSA public modulus.

So, changing the algorithm used actually requires us to go into the code more deeply than just adding one method to implement one new function. This is the motivation for using standard algorithms for RSA, DES, and Diffie-Hellman, which are the most commonly used.

The goal of code generation may be merely to simulate automatically the execution of a security protocol. From this point of view, the actual details of a Public Key Infrastructure, or of the bytes in a cipher message, are probably not the most important concerns.