

# The design and implementation of Obol

Tage Stabell-Kulø    Tåle Segtan Skogan    Per Harald Myrvang

Department of Computer Science  
University of Tromsø  
Norway

## Abstract

Obol is a special purpose programming language for the implementation of security protocols. It has been designed with the success of BAN in mind. In particular, the language mimics the representation used to express protocols before idealization.

Obol is a (very) high level language, and, in most cases, it is straight forward to convert a protocol expressed in the traditional notation (e.g. :  $A \rightarrow B : \{N_A, N_B\}_{K_{AB}}$ ) into Obol.

Obol is well suited for experimenting with security protocol design.

The Obol runtime, named Lobo, has been implemented; the implementation is presented and the design discussed.

Several well known protocols are run to justify the approach.

## 1 Introduction

“Early binding is extremely wicked.” Roger Needham

The interest in security protocols stems from the fact that it is impossible to build any non-trivial distributed systems without them. We are both interested in whether a protocol is logically correct, and in the quality of the implementation. When protocols are discussed and analyzed, they are more often than not described in a traditional message-focused notation; the following, the Yahalom protocol, is a

typical example (and taken from [10, page 30]):

*Message 1*  $A \rightarrow B : A, N_a$   
*Message 2*  $B \rightarrow S : B, \{A, N_a, N_b\}_{K_{bs}}$   
*Message 3*  $S \rightarrow A : \{B, K_{ab}, N_a, N_b\}_{K_{as}},$   
 $\{A, K_{ab}\}_{K_{bs}}$   
*Message 4*  $A \rightarrow B : \{A, K_{ab}\}_{K_{bs}}, \{N_b\}_{K_{ab}}$

The first step to ensure correctness is to analyze the protocol. In order to analyze any protocol with BAN (or any of the siblings [24]) we need to *idealize* the messages. This is a manual process. As an example, the following is the idealization of Message 4 above (also obtained from [10]):

$\{A \stackrel{K_{ab}}{\leftrightarrow} B\}_{K_{ab}}, \{(N_b, A \stackrel{K_{ab}}{\leftrightarrow} B, S \equiv (A \stackrel{K_{ab}}{\leftrightarrow} B))\}_{K_{ab}}$

Regardless of the merits of BAN, it can safely be said that the process of idealization is difficult, crucial, error prone, and, alas, impossible to prove correct. The value of the analysis hinges on the correctness of the idealization.

In a somewhat informal manner, we can say that the effort needed to idealize a protocol is the “distance” between the description and BAN. How hard it is to get the idealization correct depends on the protocol, and can be seen also as a measure of the protocols’ complexity. The more complex the protocol, the more important it is to analyze, and the more difficult the idealization (both to actually do, and to do correctly).

An implementation is taken to be correct if it does not violate any of the assumptions made as part of the analysis. Obviously, an implementation of a security protocol should be as secure as the protocol itself. In particular, when a protocol uses well-protected encryption keys, and the protocol itself has been designed to meet the highest demands for security, a considerable effort should go into the implementation.

When the idealization of a protocol has been found to achieve a suitable set of goals, implementation can get underway. The starting point for the implementation would be the protocol description. There is a substantial degree of freedom in the hands of the implementor and a wide range of implementations could all be correct even though they are incompatible. Such an incompatibility can stem from use of encryption, message formatting issues, naming, and so on. Any effort on the analytical level can be voided by an error in the implementation.

If the language of choice is Java or C, the “distance” from the protocol description to the running code is *considerable* longer than the “distance” between the description and the idealization. Also here, the greater the distance, the more likely it is for an error to be made. To that end, Obol is a special purpose programming language, designed for the sole purpose of implementing security protocols (such as the one above).

Obol is a programming language for a very specific environment. The programs implements protocols, and the runtime for the language reflects this. Very briefly we can say that clients download into the runtime, which is named Lobo, programs written in Obol (e.g. protocols). Lobo then exchanges messages with other parties, encrypts and decrypts, and deals with all the low level details. In general terms we can say that Obol tries to embed in the language itself all the aspects that are “only” engineering (such as padding and formatting), while exposing those related to analysis (such as which components go into a message).

The rest of the article is structured as follows: We first present the design goals of Obol, before we describe the design and implementation. We then give a few examples of Obol programs implementing non-trivial protocols, as well as a discussion why e.g. SSH currently cannot be programmed in Obol. An overview of related work is given, and we conclude with future work.

## 2 Design Goals

The goal of Obol is to make the “distance” from a protocol description to running code no longer than the “distance” to the idealization. We believe this has been achieved.

We wanted Obol to be a language one could apply in settings where it for some reason was undesirable to write a full protocol stack. A program in Obol does not describe a full protocol: It describes (only) the implementation at *one end*. For that reason, an Obol program is not a protocol specification.

Being based on a combination of engineering and analysis, Obol has primitives rooted in both camps. One part of the language closely follows the expressional power of the language used to describe protocols. The second part follows from the semantics of the message components exposed by an analysis in BAN. We will use the Yahalom protocol to demonstrate the different primitives.

*Message 1*  $A \rightarrow B : A, N_a$

Obviously, it must be possible to send and receive messages, and means to express both sending and receiving must be primitives in the language. An integral part of exchanging messages is to decide on a formatting convention, how to detect and identify message components, and so on.

Also an integral part of sending is a naming and addressing scheme, and (a derivative of) this scheme must be used also as message components. The design of this naming scheme raises an intriguing question on whether naming PER SE has security implications or not, and whether this is “only” implementation details. In any case, Obol supports a flexible naming scheme.

In Obol, encryption and decryption is exposed to the programmer, but details about padding and initialization vectors are not. We believe that a fruitful abstraction is to indicate which components are to be encrypted. Upon receiving, if (parts of a) packet needs to be decrypted, the Obol runtime will verify that the correct key is used. Then, as a separate issue, it is established whether the packet is the correct one. This must be done by comparing elements found inside the message after decryption with objects supplied by the programmer. This closely follows the ideas of correctness in BAN.

In this protocol  $N_a$  is a nonce, and it must be generated as part of the program execution. In addition to nonces, the implementation of many protocols also needs to generate fresh keys (both symmetric and asymmetric), and time stamps. Obol supports this.

<b>send</b>	Send data.
<b>receive</b>	Try to obtain a message, with component matching.
<b>encrypt</b>	Shared- and public key encryption.
<b>decrypt</b>	Decrypt, with component matching.
<b>generate</b>	Generate nonces and keys.
<b>believe</b>	Promote data into knowledge (or beliefs).

Table 1: Primitives in Obol.

Description:	$A \rightarrow B : \{A, K_{ab}\}_{K_{bs}}, \{N_b\}_{K_{ab}}$
Obol:	<code>(send B (encrypt Kbs A Kab) (encrypt Kab Nb))</code>
Interpretation:	$\{A \stackrel{K_{ab}}{\leftrightarrow} B\}_{K_{ab}}, \{\langle N_b, A \stackrel{K_{ab}}{\leftrightarrow} B, S \equiv (A \stackrel{K_{ab}}{\leftrightarrow} B) \rangle\}_{K_{ab}}$

Table 2: A message, its implementation and interpretation.

The correctness of the protocol does not depend on whether the first message is sent as  $\langle A, N_a \rangle$  or  $\langle N_a, A \rangle$ . When a message is sent it will be left to the runtime to determine how the message components best can be represented. We believe this is a sound separation of concerns.

*Message 2*  $B \rightarrow S : B, \{A, N_a, N_b\}_{K_{bs}}$

Encryption is integral to security protocols, as is decryption; in this example a symmetric crypto system is used. With an asymmetric system, encryption and decryption with both types of keys (encryption and decryption, signing and verification) is supported. It is not possible to reliably decrypt message without an A PRIORI agreement on the format of the message; this is part of Obol’s runtime.

When this message is received by  $S$  the naming scheme’s importance must be recognized. It is evident in the analysis that  $S$  controls the creation of a shared key between  $B$  and  $A$ , and is thus in some way able to deduce whom  $A$  is. We will return to this later. The messages 3 and 4 does not add anything to our list of features in the language, but will be discussed below in a different context.

To sum up, the parts of the core of Obol derived from the language used to describe messages are primitives for sending and receiving messages, encryption and decryption, machinery to deal with a naming scheme, and the ability to generate random material for nonces and keys.

The process of idealization exposes a different set of properties hidden in the protocol description. Con-

sider the first part of the third message which is received by  $A$ :

$$\{B, K_{ab}, N_a, N_b\}_{K_{as}}$$

The security of a system using this protocol depends not only on what the message contains, but also on whether  $A$  is able to reliably determine what is inside. One of the overarching principles on engineering security protocols [1, 2, 3, 12] is concerned with the content of messages: Every message should explicitly say what it means; the interpretation of the message should depend only on its content, and it should not be necessary to use any other context. After decryption and verification that the decryption was correct,  $A$  must locate the four components, which have three different types: a name, a shared key, and two nonces, one of which should be identifiable as  $A$ ’s own. The ability to identify the components, and assign them their correct type, is crucial for upholding the assumptions made during the analysis, and thus for security. In particular, the above message must be discarded unless  $N_a$  can be found within it; the freshness of  $K_{ab}$  hinges solely on the ability to establish this fact. Since Obol supports such “look ahead”, “Fail-Stop” [15] protocols can be efficiently implemented. The only means to receive a message in Obol is to give a specification of the components that are to be found in it. In the `receive` statement one can also specify components expected to be found within encrypted parts.

Along the same line of argument, if the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal’s name explicitly in the message and not rely on the channel the message was delivered on to acquire this knowl-

edge. For this reason, the notion of a communication channel can not be expressed in Obol. When analyzing protocols it is not relevant *how* a message arrived, and in Obol it is left to the runtime to search for messages that matches the requirements of the application.

After a message has been received, we know from the axioms in BAN, and more explicit in GNY [14], that data found in messages can be promoted into belief: Bitstrings are promoted into encryption keys. This is a decision crucial for security, and in Obol no promotion of this kind can occur without doing it explicitly.

We now turn to the second part of the third message:

$$\{A, K_{ab}\}_{K_{bs}}$$

This component can not be read at all by  $A$  because it is encrypted by a key he does not have access to. For this reason, Obol must have primitives to deal with message components of unknown type; these are said to be of the *anonymous* type. In practice, one of two possible actions are reasonable when dealing with the content of a anonymous variable: Either forward it to another party (as is done here), or make some assumption on the content and turn it into a known type (such as a key). The latter is what  $B$  will have to do; how this is done in Obol is shown on line 16 of the Obol program that implements  $B$ 's part in this protocol (see Section 4.2).

Taken together, the core of Obol consists of a small set of primitive operations aimed at the core functionality of security protocols. The operations are summarized in Table 1. The operators are designed to facilitate the *implementation* of security protocols, and are designed to ease the translation from a protocol description that will also be used as a starting point for an idealization. Table 2 shows Message 4 from the Yahalom protocol, its idealization (taken from [10]), and the implementation in Obol. Please note that that the snippet of code shown in Table 2 can be run as shown; a full listing of the protocol implementation is included in Section 4.2.

### 3 Design and Implementation

In the previous section we discussed the goals of Obol, here we will discuss the design and implementation.

#### 3.1 Design

The three basic parts of a programming language are expressions, statements, and declarations [22]. Following the style in which protocols are described, we want Obol programs to consist of a sequence of expressions, not statements, because expressions make it easy to write short, nested code as is well documented by the Lisp experience [13]. Ideally, we would like Obol to be a language consisting of only expressions without side effects; i.e. a functional language [22]. It can be argued that such languages encourages readability and correctness [4]. However, we certainly wanted to have variables, but Obol still has a functional “taste”; the functional programming style removes the need for temporal variables.

We have chosen dynamic typing. The values have type, not the variables themselves. Variables are created dynamically and have global scope. Values of anonymous type are kept in variables prefixed with  $*$  and referred to as anonymous variables.

The order of evaluation is strictly from left to right. Statements can be nested, and evaluation is from inner towards outer. Again, this is also shown in the statement in Table 2, which contains three statements, two of them embedded in an outer statement.

For reasons of simplicity, and to avoid precedence rules, Obol is designed to have a prefix notation, i.e. all expressions start with an operator, followed by zero or more arguments. A syntactically valid Obol statement was shown in Table 2.

We will now discuss the particularities of the six primitives in Obol (shown in Table 1).

**1: Believe** In Obol data can be “believed” to have structure; this is how a bitstring is promoted into an encryption key. We model the `believe` primitive after the corresponding one in BAN. We use it to make assumptions (this key is

“good”), and to promote data that has been received. Embedded in the primitive is also declaration of variables. The syntax is `(believe variable value type attribute)`, and a typical example will be `(believe K 0x1234 shared-key ((alg AES)))`; the runtime will verify that the value (here a number) is suitable for promotion into a key.

**2: Generate** When nonces, timestamps and keys (both types) are needed they must be explicitly generated. The syntax is `(generate type attributes)`. Generate is used with `believe` to create variables of some specific type: `(believe K (generate shared-key ((alg AES) (size 128))))`.

**3: Encrypt** How data is encrypted depends on the type of the key; the runtime will dispatch to the correct routines. The syntax is `(encrypt key data attributes)`. Keys must explicitly have been believed to be good. For example

```
(believe K (generate shared-key ((alg AES)
                                  (size 128))))
(encrypt K "secret")
```

**4: Decrypt** As discussed earlier, decrypting first and foremost requires that the decryption succeeds; this must be determined by verifying redundant information inside the package (such as a checksum). Then the decrypted content must be examined to decide if it matches the protocol’s requirements. The syntax is `(decrypt key ciphertext pattern attributes)`. How this pattern matching is used will be demonstrated in a moment.

**5: Send** The sending of messages are central to any protocol. In Obol, the syntax is simple: `(send receiver data)`. The data will usually be a list of variables (or functions that return an object). For example `(send "host:port" A (encrypt K Na))` where the second component of the message would be `Na` encrypted with `K`.

We have chosen a simple ASCII format for messages. Each message element is encoded in hex and surrounded by parenthesis and the message as whole is again surrounded by parenthesis. We chose an

ASCII based format to make debugging and inspection of the messages simple. The nested parenthesis structure makes parsing easy. Here is a message containing a 4 byte timestamp and a 128 bit AES key:

```
((c3c249ae)(1c7dc3fd915199830a7c1959e7aa9d1f))
```

Notice that the message encoding contains no information on how to decode the individual message elements; the message elements behave like bit sequences with no inherent meaning. This is sufficient because the receiver must have the necessary information to decode the message; decoding happens in the context of a pattern such as in a receive statement. If the first element in the pattern is a nonce, the hex encoded bit sequence in the message element is interpreted as a nonce and if the pattern element is a string it is decoded as a string, and so on.

Which format is chosen, as long as it is based on sound engineering principles, should be irrelevant.

**6: Receive** Receive is by far the most complex primitive. Since communication channels PER SE cannot be expressed in Obol, message receipt cannot be determined by where or how a message appears. Instead, a message receipt must be described in terms of expected content. The syntax is `(receive pattern attributes)`.

Take as example the fourth message of the Yahalom protocol:  $A \rightarrow B : \{A, K_{ab}\}_{K_{bs}}, \{N_b\}_{K_{ab}}$ . The implementation of this statement is as follows (the line numbering is not part of the implementation):

```
1 (receive *1 *2)
2 (decrypt Kbs *1 A *Kab)
3 (believe Kab *Kab shared-key ((alg AES)
                                  (size 128)))
4 (decrypt Kab *2 Nb)
```

1. A message is expected that can be parsed into two separate components; these are stored in two anonymous variables as a side effect. The expression returns the message.
2. The first component should be decryptable using the key stored in `Kbs`, and inside it must again be possible to identify two components, of which the first must match what is currently

stored in the variable `A`. Only if both are true does the call succeed. The second component found after decryption is assigned to the anonymous variable `*Kab`.

3. The content of the variable `*Kab` is promoted to a shared key and stored in the variable `Kab`.
4. It is verified that the contents of `Nb` can be found in the anonymous variables by decrypting with the key stored in `Kab`.

In addition to these six core primitives, there is also support for generating and using Diffie-Hellman type keys, both for key-exchange schemes and for reading and writing such keys to storage.

If we wanted to be able to write protocols that are compatible with existing implementations, Obol would have to be Turing complete. For example, an implementation can add a transformation of data to a packet (such as a checksum). Obol is not Turing complete because we wanted to investigate how much flexibility we could achieve while retaining the high-level aspects of the language.

### 3.2 Lobo

We have implemented a prototype runtime for Obol; this runtime is called Lobo. The runtime implements Obol with an interpreter, which is designed in the standard way: It takes an expression and an environment as input, evaluates the expression in that environment and returns the result. Since a top-level expression may contain nested Obol expressions, a recursive structure was natural for the evaluation part. Because we designed Obol expressions to always begin with an operator, the core of the evaluation was very easy to implement. The complete runtime was implemented in about 2500 lines of Common Lisp.

The runtime supports multiple Obol scripts in parallel, and has an API which applications can use to load scripts into Lobo and get results back again. The application gets a handle to each script it loads and uses this handle to communicate with the script instance.

If the application doesn't run in the same address space as Lobo, the handle will typically hide a proxy. The application will call an initialization method on

the proxy which behind the scenes sets up a secure channel to Lobo and offers a method for loading scripts when ready. After loading a script, the application waits for Lobo to return a value to the proxy, and the proxy may offer other functionality like a close method. All communication between Lobo and the proxy must happen according to some well-defined protocol that handles issues like marshalling of values. The exact nature of the proxy and the corresponding glue in the runtime will depend on the implementation languages; a Lobo runtime implemented in Java servicing Java applications will be different from a runtime implemented in Common Lisp and servicing Python applications. We have implemented a proof-of-concept protocol with a Common Lisp proxy and corresponding glue in the runtime.

To handle multiple scripts in parallel we designed Lobo as an event based system with a main event queue and handler functions for the various event types. Each script runs until blocked, e.g. by a receive with no matching messages in the queue. Control is then yielded to the next active script.

Performance has not been a goal. Rather, we were interested in finding suitable abstractions for writing security protocols. However, we believe that performance will not be a problem in most cases for the following reasons: Security protocols are typically short with few messages, and spends most of their time either waiting for the network, or doing "low level" cryptographic operations. We can do little with the network latency and low level crypto can be handled by hardware accelerators or by hand-tailored code.

## 4 Examples

Below are some examples of protocols implemented in Obol; they all run on the current implementation of Lobo.

### 4.1 Ping-Pong protocol

Our first example, is a trivial protocol to verify the presence of another party (expressed in the custom-

ary notation):

Message 1:  $A \rightarrow B \quad A, B, N_A$   
Message 2:  $B \rightarrow A \quad B, A, \{N_A, N_B\}_{K_{AB}}$   
Message 3:  $A \rightarrow B \quad A, B, N_B$

Under the assumption that the key  $K_{AB}$  is “good” for communication between Alice and Bob, the exchange of encrypted, fresh random numbers as shown will make both parties believe that the other party is online *now*.

This protocol can be programmed in Obol<sup>1</sup> and a client would transfer the program (no more than a few hundred bytes in all) to Lobo.

In this example the communication is initiated by Alice, and she executes the following program (the line numbers are not part of the program):

```
1 (believe $L "myhost:port" address)
2 (believe $R "hishost:port" address)
3 (believe $K (load "K.key") shared-key
   (alg AES) (size 128))
4 (believe $N (generate nonce ((size 128))))
5 (send $R $L $R $N)
6 (receive $R $L *1)
7 (decrypt $K *1 $N *2)
8 (send $R $L $R *2)
9 (return t)
```

This program is represented as a string, and is transferred to Lobo over some (secure) communication channel (a system-call channel, for example). The numbers below refer to the line numbers in the source code:

1. In this program, variables are here identified by having ‘\$’ as first character.

The semantics of the keyword “address” is that data held by these variables are interpreted to be addresses of some sort; i.e. the *type* of the value referred to by the variable \$L is “address”.

The net effect of this statement is that the variable \$L is created in the namespace, and set to (point to) information about the local system.

2. The variable \$R is created, and set to information about the Bob.
3. The variable \$K is given type “shared-key”, and loaded from a file named “K.key”.

---

<sup>1</sup>One could say that the protocol is *Obolable*.

Notice that the programmer has stated two assumptions he is making about the key (the algorithm where it is to be used, and it’s length). Lobo can now verify that they hold. In most cases, which algorithm is used is not important, but for some reason the programmer wants AES in this protocol.

4. The nonce is generated and bound to the variable \$N.
5. A message (containing the set of values held by the variables \$L, \$R, and \$N) is sent to the value of \$R.
6. The program blocks until a message is received, presumably from a channel assumed to originate with \$R (a TCP connection established as a side effect of the previous statement, for example).  
The message is received (by the program) when it can be parsed by the runtime into three components where the first must be found to be equal to \$R and the second to \$L; the third can be anything and is assigned to the anonymous (typeless) variable \*1.
7. The value of the anonymous variable \*1 must be decryptable with the key \$K. One of the components after decryption must be \$N. The other one is unknown (unrecognizable), and is assigned to the anonymous (typeless) variable \*2.
8. A new message is sent to \$R; the unknown message component held in the variable \*2 is made part of the message.
9. The return value is “true”, informing the caller that the protocol executed to completion; how to return a value will obviously depend on the implementation. If Lobo fails to execute the Obol program, an environment-specific indication that an error has occurred will be returned (e.g. a Java Exception).

When Alice is returned the value “true” she can draw the conclusion that if it is (still) true that the key identified by \$A in her key repository is shared only with Bob, Bob is online (now).

## 4.2 Yahalom

In Section 1 we used the Yahalom protocol to explain the rationale for Obol. Following is the Obol

implementation of this protocol (line numbers are not part of the implementation):

```

;; Implementation of A
;;
1 (believe A "host-A:9000" address)
2 (believe B "host-B:9000" address)
3 (believe Kas (load A) shared-key ((alg AES)
   (size 128)))
4 (believe Na (generate nonce ((size 128))))
  ;; 1: A->B: A, Na
6 (send B A Na)
  ;; 3: S->A: {B, Kab, Na, Nb}Kas, {A, Kab}Kbs
7 (receive *toA *toB)
8 (decrypt Kas *toA B *Kab Na *Nb)
9 (believe Kab *Kab shared-key ((alg AES)
   (size 128)))
10 (believe Nb *Nb nonce)
  ;; 4: A->B: {A, Kab}Kbs, {Nb}Kab
11 (send B *toB (encrypt Kab Nb))

;; Implementation of B
;;
1 (believe B "host-B:9000" address)
2 (believe S "host-S:9000" address)
3 (believe Kbs (load B) shared-key ((alg AES)
   (size 128)))
  ;; 1: A->B : A, Na
4 (receive *A *Na)
5 (believe A *A address)
6 (believe Na *Na nonce)
  ;; 2: B->S: B, {A, Na, Nb}Kbs
7 (believe Nb (generate nonce ((size 128))))
8 (send S B (encrypt Kbs A Na Nb))
9 ;; Message 4 A -> B : {A, Kab}Kbs, {Nb}Kab
10 (receive *1 *2)
11 (decrypt Kbs *1 A *Kab)
12 (believe Kab *Kab shared-key ((alg AES)
   (size 128)))
13 (decrypt Kab *2 Nb)

;; Implementation of S
;;
  ;; 2: B->S: B, {A, Na, Nb}Kbs
1 (receive *B *fromB)
2 (believe B *B address)
3 (believe Kbs (load B) shared-key ((alg AES)
   (size 128)))
4 (decrypt Kbs *fromB *A *Na *Nb)
5 (believe A *A address)
  ;; Se comment below
6 (believe Na *Na nonce)
7 (believe Nb *Nb nonce)
  ;; 3: S->A : {B, Kab, Na, Nb}Kas, {A, Kab}Kbs
8 (believe Kas (load A) shared-key ((alg AES)
   (size 128)))

```

```

9 (believe Kab (generate shared-key ((alg AES)
   (size 128))))
10 (send A (encrypt Kas B Kab Na Nb)
    (encrypt Kbs A Kab))

```

Notice in the implementation of  $S$  that we have chosen to promote the two components into nonces (lines 10 and 11). This is not strictly necessary, as they could have been included in the encryption expression in the line 15 without having been promoted. However, by promoting them we give the runtime the possibility to examine them as nonces (and not just data). This can be used to verify that they haven't been seen before, that they seem to be random, and so on. This illustrates how one with ease can experiment with protocols and their implementation when the protocol has been implemented in Obol.

### 4.3 SSH

SSH is a widely used security system and it is natural to ask if Obol can be used to implement (parts of) it. The answer is basically “no”; we discuss the reason below.

The complexity of SSH stems from several sources: The runtime carries much state; for instance, some messages will not be legal before the user has been authenticated, and a re-keying operation must not be started if there is already one going on. There is poor support for this in Obol.

Another complicating factor is that SSH is really a whole architecture composed of multiple protocols working on several levels. A third complicating factor is that there are states which have more than one legal next state. This complicates receiving messages, and requires conditionals to be introduced in Obol.

Yet another obstacle is the re-keying sub-protocol; after maximum one hour a new session key is established by running part of the original protocol anew. But this time the protocol is run on top of the protocol multiplexing machinery that has been established.

As a whole, SSH is a suite of protocols. Obol in its current form is not intended for such protocols. Rather it is targeted at purely security related functionality, especially authentication protocols. How-

ever, we believe it would be possible to use Obol when implementing an application that provided the same functionality as say a SSH client. But such a solution would not be binary compatible with SSH.

#### 4.4 EKE

EKE (Encrypted Key Exchange) was proposed by Bellare and Merritt [5] to secure password-based protocols against dictionary attacks. It also achieves perfect forward secrecy, that is, the disclosure of the password (long term secret) does not compromise the session key produced by the protocol.

Let  $A$  and  $B$  be the two principals in the protocol,  $P$  a shared long term secret (password or secret key), and  $K_t$  the public part of a temporary public key pair. The protocol consists of five messages where the last three are for mutual verification of the key; we present only the two first which show the essence of the protocol:

*Message 1*  $A \rightarrow B : A, \{K_t\}_P$   
*Message 2*  $B \rightarrow A : \{\{K_{ab}\}_{K_t}\}_P$

Note that the perfect forward secrecy comes from the fact that the session key is encrypted with a temporary public key. Even if the long term secret  $P$  is compromised later, this will only reveal  $K_t$ ; the attacker cannot get to the session key  $K_{AB}$  because it is still encrypted with  $K_t$  and the corresponding private key needed for decryption has hopefully been destroyed.

```
;; Implementation at A
;;
;; Message 1 A->B : A, {Kt}P
1 (believe A "host-A:9000" address)
2 (believe B "host-B:9000" address)
3 (believe P (load "P.key") shared-key
  ((alg AES)))
4 (believe (Kt Kt-1) (generate public-key
  ((alg RSA) (size 512))))
5 (send B A (encrypt P Kt))
6 (believe Kab (decrypt Kt-1 (decrypt P
  (receive *1))) shared-key ((alg AES)))

;; Implementation of B
;;
;; Message 2 B->A : {{Kab}Kt}P
1 (believe P (load "P.key") shared-key
```

```
((alg AES)))
2 (receive *1 *2)
3 (believe A *1 address)
4 (believe Kt (decrypt P *2) public-key
  ((alg RSA) (keytype RSAPublicKey)))
5 (believe Kab (generate shared-key ((alg AES)
  (size 128))))
6 (send A (encrypt P (encrypt Kt Kab)))
```

## 5 Related Work

Other projects that are relevant to Obol can be divided into four different categories depending on their approach: Logics, Formal Description Techniques, languages that are compiled, and middleware solutions.

### 5.1 Logic

Various methods can be used to analyze protocols to determine whether they achieve their goals, in the hope that the designers can convince themselves that the protocol's assumptions, state transitions and desirable goals are sound and attainable [19]. These analysis tools have also successfully been used to find and prove design flaws in existing protocols, often in protocols that were believed to be sound [20]. There are several classes of tools available for such analyses, ranging from modal logics [10, 14, 26], to Higher Order Logics rewrite systems [9], and to state attainability deducers [19, 20]. Common to all these approaches is that they prove or disprove the reachability of a protocol's goal state from its initial assumptions via a set of transitions, and that they operate on a description of protocols that is not executable.

Although Obol has its roots in these logics, being a programming language sets it apart.

### 5.2 Formal Description Techniques

LOTOS (Language Of Temporal Ordering Specification) was developed by OSI in the late eighties as a Formal Description Technique [7, 17]. It is a language for the description of protocols. It has complementary formalisms for 'data' based on ACT

ONE [11] and 'control' based on CSP [16]. The language has formally defined syntax, static semantics defined by an attributed grammar and dynamic semantics described in terms of inference rules. The main goal of the language is to describe protocols in such a way that reasoning about their correctness is possible. Such reasoning is done by investigating the possible states the protocol can be in, depending on the messages that are received. In general, this model has an infinite number of states, where time is added to the usual state machine model via a special "time passage" action. Estrel is one of the family of languages used to describe real-time systems using the state machine model [8, 6].

This approach is fundamentally different from the one taken in Obol. A protocol implemented in Obol is not a vehicle for analysis.

Another approach is to assume that TCP/IP is available, and then place a new layer between the application and the transport layer [21]. This new layer is then responsible for negotiating security properties between the parties. The system, named LEI (Logical Element of Implementation) can interpret and implement any security protocol from its specification. As is custom for these approaches, there is only one protocol description and both sides need to run identical software. In some sense this approach could be classified as middleware.

### 5.3 Implementation languages

Prolac is a language for implementation of protocols [18]. The language can be compiled into C, and, as is the case with Obol, both sides need not be implemented in the language. Prolac is geared towards the implementation of "traditional" protocols and do not have any special features for the implementation of security protocols.

Prolac compiles to C and the idea is that the resulting code can be embedded in an application or system. In this respect, Obol has a different goal since we want the protocol to be executable at once. In particular, we can not modify the hosting system to link with a new version of a protocol.

### 5.4 Middleware

"Da Capo++" is a middleware system where many of the application's needs and communication demands can be specified in terms of QoS values [23]. Da CaPo++ has a well defined protocol machinery (named Lift) which is at the core of the system. The data is managed by the Lift, and passed to modules that are inserted (or removed) according to the QoS specifications of the application. There is a rich API to enable applications to compose the system to its needs. The performance is good.

Da CaPo++ also places security under the same QoS regime as other resources available to the system. The means that the "degree of privacy" has to be specified as a QoS value (a number). The problem is, obviously, that any number needs a semantic mapping to be useful. In Da CaPo++ this mapping has been constructed in the following way [23, Section E.2]: The application can specify the required strength of the cryptographic algorithm that is to be applied to the data. This strength is derived from how long the application wants the data to be protected as a function of who a likely attacker might be, measured in money. For example, an application can require that the data must withstand an attack by a "determined group" or "competing enterprise" for a number of years, and the value of the information. Based on this, the system must provide security mechanisms whose breaking cost are higher than this value. In addition, the user has to specify how likely these promises should be met. The authors of [23] notes that it is easier for the user than for the system to add "quality" to data, and to perform the mapping required for the system to find a suitable algorithm.

The system will maintain a database of algorithms (and their implementations), and by using the infrastructure provided by the system, a suitable protocol can be constructed (by inserting in the Lift the appropriate modules). This design relieves the application from having to specify which cryptographic algorithm to use, as well as being flexible; new algorithms can be installed at any time, without having to change the applications.

Da CaPo++ and Obol are very different systems. Da CaPo++ is *one* system that is intended to run on both sides of a communication channel; it is middleware. Obol, on the other hand, is a subsystem for protocol execution, and a protocol realized with

a program written in Obol can communicate just as well with a system that does not run Obol.

Even though the systems are very different, there are two aspects where Da CaPo++ is relevant (for Obol):

- Da CaPo++ views security as one of many QoS attributes, and there are mechanisms to manipulate security attributes in the same manner as one manipulates other relevant aspects of the system. If Obol was integrated into Da CaPo++ the resulting system would be even more flexible than what Da CaPo is today. Rather than mainly altering encryption algorithms, Da CaPo++ could also freely change the protocol.
- In general Obol programs do not carry instructions for the run time on which algorithms to choose, but this could conveniently be done by the mechanisms offered by Da CaPo++.

## 6 Future Work

The first implementation of Lobo was a prototype to demonstrate the viability of the approach; all the examples shown here have been run on it. This first prototype was done in Common Lisp. We chose a format for messages rather arbitrarily. Our format is native to our implementation, and effectively blocks any interoperability with existing systems. We aim at introducing a language for the formatting of messages. Since messages often hold general transformations of the data (such as checksums) the language needs to be Turing-complete. Ideas for how this can be done has been taken from the format form in Common Lisp.

As part of the Arctic Beans project, we want to make programmable protocols available to components in the reflective middleware system OOPP. This implementation of Lobo has been written in Python; it supports a somewhat different set of primitives.

Also as part of Arctic Beans, work is underway to make Obol available to components running in EJB; this would offer more flexibility than use of the interceptor mechanism. This implementation is written in Java.

## References

- [1] M. Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.
- [2] R. Anderson and R. Needham. Robustness principles for public key protocols. In *Proceedings of the International Conference on Advances in Cryptology (CRYPTO 95)*, 1995.
- [3] R. Anderson and R. Needham. Programming satan’s computer. In J. van Leeuwen, editor, *Computer Science Today - Recent Trends and Developments*, number 1000 in Lecture Notes in Computer Science, pages 426–440. Springer-Verlag, 1996.
- [4] J. Backus. Can programmers be liberated from the van Neumann style? A functional style and its algebra of programs. In R. L. Ashenurst and S. Graham, editors, *ACM Turing Award Lectures - The first Twenty Years*. ACM Press, 1977.
- [5] S.M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. *Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland*, 1992.
- [6] G. Berry and G. Gonthier. The ESTREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 2(19), 1992.
- [7] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. In Peter H. J. van Eijk, Chris A. Visser, and Michel Diaz, editors, *The formal description technique LOTOS*, pages 23–73. North-Holland, 1989.
- [8] Frédéric Boussinot and Robert de Simone. The ESTEREL language. *IEEE Transactions on Software Engineering*, 9(79):1293–1304, September 1991.
- [9] Stephen H. Brickin. Automatically detecting most vulnerabilities in cryptographic protocols. DARPA Information Survivability Conference and Exposition, Hilton Head Island, SC, USA, jan 2000.
- [10] Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. *ACM*

- Transactions on Computer Systems*, 8(1):18–36, February 1990.
- [11] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985.
- [12] N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley & Sons Inc., 2003.
- [13] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 2nd edition, 2001.
- [14] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning about Belief in Cryptographic Protocols. In *Proceedings of the IEEE 1990 Symposium on Security and Privacy*, pages 234–248, Oakland, California, May 1990.
- [15] Li Gong and Paul Syverson. Fail-Stop Protocols: An Approach to Designing Secure Protocols. In *Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications*, Urbana-Champaign, Illinois, September 1995.
- [16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [17] ISO. Information processing systems — Open systems interconnection — Estelle — a formal description technique based on an extended state transition model, 1989.
- [18] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the Prolac protocol language. In *ACM SIGCOMM*, pages 3–13, 1999.
- [19] Catherine Meadows. Formal Verification of Cryptographic Protocols: A Survey. In *Advances in Cryptology – Asiacrypt ’9*, volume 917 of *Lecture Notes in Computer Science*, pages 133–150. Springer-Verlag, 1995.
- [20] Catherine Meadows. The NRL Protocol Analyzer: An Overview. *The Journal of Logic Programming*, 26(2):113–131, February 1996.
- [21] L. Mengual, N. Barcia, E. Jiménez, E. Menasalvas, J. Setién, and J. Yágüez. Automatic implementation system of security protocols based on formal description techniques. In Antonio Corradi and Mahmoud Daneshmand, editors, *Proceedings of the Seventh IEEE Symposium on Computers and Communications*, pages 355–360. IEEE Computer Society, July 2002.
- [22] J. C. Mitchell. *Concepts in programming languages*. Cambridge University Press, 2003.
- [23] Burkhard Stiller, Christina Class, Marcel Waldvogel, Germano Caronni, Daniel Bauer, and Bernhard Plattner. A flexible middleware for multimedia communication: Design implementation, and experience. *IEEE JSAC: Special Issue on Middleware*, 17(9):1614–1631, 1999.
- [24] Paul Syverson and Iliano Cervesato. The logic of authentication protocols. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design: Tutorial Lectures*, volume 2171 of *Springer Lecture Notes in Computer Science*, pages 63–137. Springer Verlag, 2001.
- [25] Paul F. Syverson and Paul C. van Oorschot. On unifying some cryptographic protocol logics. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 14–28, Los Alamitos, California, USA, May 1994. IEEE Computer Society Press.
- [26] Paul F. Syverson and Paul C. van Oorschot. A unified cryptographic protocol logic. CHACS Report 5540-227, Naval Research Laboratory, Washington, USA, 1996. Parts of this paper appeared in preliminary form in [27] and [25].
- [27] Paul C. van Oorschot. Extending cryptographic logics of beliefs to key agreement protocols (extended abstract). In *Proceedings of the First ACM Conference on Computer and Communication Security*, pages 232–243, November 1993.