Cordial Security Protocol Programming
The Obol Protocol Language

Per Harald Myrvang[1] and Tage Stabell-Kulø[2]

[1] Bodø Graduate School of Business, University College of Bodø, Norway
permyr@gmail.com
[2] Department of Computer Science, University of Tromsø, Norway

Abstract. *Obol* is a protocol programming language. The language is domain specific, and has been designed to facilitate error-free implementation of security protocols.

Selecting the primitives of the language is, basically, concerned with determining which issues needs to be visible to the protocol programmer, and which can be left to the runtime without further ado.

The basic abstractions of Obol has been modelled after the ones offered by the BAN logic of authentication. By building on these abstractions Obol makes it less hard to bridge the gap between logical analysis and implementation.

Obol has been designed with the implementation of security protocols in mind, but the language can be used to implement also other types of protocols.

At the core of the design and implementation is pattern-matching machinery enabling the runtime to parse packets as they arrive in order to free the programmer from a wide range of low-level issues know to foster all sorts of implementation difficulties.

1 Introduction

The interest in security protocols stems from the fact that it is impossible to build any non-trivial distributed systems without them. Unfortunately, due to their harsh operating environment, they are notoriously difficult to design [1]. The problem partly stems from the difficulty of establishing a firm foundation for expressing the goals the protocols strives to achieve.

Implementation of these protocols is a different matter all together. There are three main issues: Traditional implementations issues, assuring that security assumptions holds, and difficulties related to cryptography. We will now discuss them in turn.

- In the context of security protocols, any seemingly innocent implementation error can render the whole endeavor useless. This means that implementation efforts must be undertaken with particular rigor. However obvious this might seem to be, there are abundant examples of programming errors leading to all sorts of problems. Also, since there is a substantial degree of freedom in the

B. Christianson et al. (Eds.): Security Protocols 2006, LNCS 5087, pp. 62–84, 2009.

hands of the implementor, a wide range of implementations could all be correct even though they are incompatible – by using different cryptographic primitives, message representation, naming schemes, error handling, and so on.
- A typical security assumption is that all (honest) participants have synchronized clocks, a task of considerable complexity [2]. This assumption can be used by including a time stamp in messages, to determine when a certificate is deemed invalid, and so on. Another common security assumption is the availability of an infrastructure to support the use of public keys (a PKI). Also this is a task of considerable complexity, see *e.g.* [3]. This type of assumptions are not related to the implementation of protocols themselves, but it would be an advantage if as many such assumptions as possible were visible to those conducting the implementation. Furthermore, using protocols in a computer system implies that it needs to be implemented in some programming language. This makes the compiler (or interpreter) part of the trusted computing base (TCB), together with supporting libraries, the operating system with it's dynamic runtime, and what not [4,5,6]. The alternative, to build all necessary software components (compilers, libraries, runtime, and so on) from scratch, is for most projects not a realistic option. Thus, most implementations must strike a balance between implementation complexity, desire for powerful assumptions, and confidence in the correctness of the implementation when it ready for deployment.
- The last family of issues are the application of cryptographic primitives. Even when the primitives themselves are correctly implemented, applying them correctly is a challenge in itself [7]. These results are not only of academic interest, as some high profile cases demonstrates [8]. Application of cryptographic primitives are hard to get right also at a higher level of abstraction [9].

It is also problematic that, in general, a design "freezes" the world view of the system. The world is dynamic, users' view of the world changes, but a computer system is more often than not a fixpoint around which users invent solutions to make the system do what they need it to do now [10]. In the security domain this is troublesome as the design and implementation of security features is funded on a careful evaluation of how the can be expected to be used. It is an inherit security problem that system can not (easily) be modified in a controlled manner.

The question is then: How can a system that contains protocols change and adapt to new operating environments? In particular, how can a protocol be changed? There can be many reasons for changing a protocol, from emergencies to "just" non-common events such as (temporal) failure of a PKI. In more concrete terms: Because implementing security protocols is a major undertaking it is unlikely that it will be changed unless it is hopelessly outdated, although it would be an advantage not having to wait quite that long. On the other hand, as the computing milieu diversifies it seems reasonable to arrange for flexibility also in the protocol domain. To this end we can only hope for a looser coupling between design and implementation in that it must neither be an Herculean task to alter a protocol that has been deployed nor replace the protocol altogether.

The rest of the article is structured as follows. In Section 2 we discuss what primitives and abstractions a security-protocol programming language should have and support. This discussion is taken forward, in Section 3, where we discuss the design choices underpinning Obol and it's runtime Lobo. Three different implementations are presented in Section 4, while several applications are discussed in Section 5. Section 6 is concerned with related work and other approaches similar to ours. Some conclusions are offered in Section 7.

2 Framework

It is hardly lack of understanding in the scientific and engineering community that is causing problems within the realm of security protocols [9,11,12]. In our view the challenge is to tackle the considerable "gap" between the manner in which protocols are presented, discussed and analysed, and the programming environment that are used to implement them.

All programming constructs are exercises in abstraction. The design choices should ensure that the essential parts of the target problem domain can be expressed and manipulated. That which is irrelevant should be dealt with by the runtime.

Below we will discuss the rationale behind the functional and non-functional requirements placed on Obol.

2.1 Non-functional Requirements

It has been suggested, although for a different type of protocols, that carefully selected abstractions is a possible solution [13]. By constructing a domain-specific language, designers are forced to adhere to established engineering techniques, resulting in language abstractions that prevent obvious mistakes, while still being useful for expressing new ideas and concepts. The first question is whether to offer suitable abstractions as libraries in an existing language (such as Java), or design and implement a language proper. Because we do not believe a system-wide view is the correct one, we can not assume that both sides of the protocol is implemented in the same language. Thus we believe Obol should be a programming language proper, for which several implementations can exist.

Because our language is targeted at implementation of security protocols, the design must be based on experience from the implementation and analysis of protocols. The world view taken by BAN can best be seen by examining the notation (symbols) and postulates. The symbols captures *what* in the realm of protocols that is deemed to be of interest (what one can talk about), and the postulates *how* these symbols are related to each other (what one can say). The abstractions offered by BAN has proven themselves to be successful and we take this as our starting point.

Security. In general, determining whether a protocol is secure or not, is undecidable. Even if only finite protocols are considered, and restrictions are placed on the generation of nonces and encryption keys, security is still undecidable [14].

The logics for authentication all make it quite clear that secrecy is a property that can not be analyzed; for a discussion see [15]. From this we conclude that because we are aiming for a versatile programming language, it will be possible to write unsecure protocols in it. In particular, the language will per se not be a tool for the verification or analysis of protocols.

Compatibility. As there's no pretence of a global authority, we can't impose our implementation on all possible participants. Thus our language cannot be based on primitives that has a whole-system view of protocols, and so our approach must deal with protocol endpoints.

We note that there seems to be two different approaches to implementing protocols: implementing existing protocols, and implementing (existing) protocols compatible with existing implementations. The latter seems harder, since it inherently includes compatibility with the implemented protocol's runtime. Because we are concerned with finding the best abstractions to offer to protocol designers, compatibility with existing protocols is not an issue here. In particular: Protocols implemented in general purpose languages (*e.g.* C or Java) can carry solutions we have explicitly left out, preventing interplay, even if the same *functionality* can be supported. We believe that by choosing abstractions with prudence we will have a programming language that can implement all relevant security protocols; this is the same line of arguments as in [13].

Adaptability. Distributed systems are faced with an ever increasing pressure to be adaptable to changes, *e.g.* changes of topology, in the range of equipment, cryptographic technology, and operating systems. The ability to accommodate change will be a necessity; one could say that protocols are too important to be hard-coded into the system. It is not so interesting to discuss what can make it desirable, or necessary, to change a protocols at run time, but a single example should suffice: The decision between public- or shared-key encryption for authentication should not be taken at design time.

Software aging [16] is particularly damaging when it comes to protocols because a change at one place must be reflected throughout the system. And, as we know, protocol design is notoriously difficult and should be avoided if at all possible. If protocols were malleable fewer wide-ranging changes would be necessary.

2.2 Functional Requirements

In deciding what functionality and which abstractions to support in the language we must study the computing model that is to underpin the programming environment.

Cryptography. When discussing the possible design choices, we allow ourself to make the sweeping assumption of perfect cryptography [17,18]. That is: We assume that good engineering can ensure the standard assumptions about encryption holds regardless of which data that is encrypted. Notice that a digital signature in this context is regarded as "encryption" because it produces a result that can not be carried on to other data.

Relevant issues. It must be possible to express *what* to encrypt, *i.e.* indicate which part of a message must be encrypted and which can be sent in clear text. This also applies to signatures and verification, where one also have to deal with computed message parts (*i.e.* message parts not sent).

Which kind of technology also has security implication; *e.g.* whether to use a public or a shared key seems to be crucial. This is not just a parameterization issue, as becomes apparent when asking how a program should deal with the case of moving from public-key to shared-key technology, such as when *e.g.* a PKI becomes unavailable: the protocol must be redesigned to cope with such a change. This in contrast to a change in *e.g.* the cipher mode or padding scheme.

Irrelevant issues. Whether a protocol applies IDEA or AES is irrelevant (from a security point of view). The same goes for issues such as padding and chaining. Such low-level issues are mostly relevant with regard to compatibility between implementations. We do not claim that it is wise to ignore many years of engineering experience and wisdom, so the actual choice of *e.g.* padding or cipher algorithm should be sound, engineering wise, at least. That does *not* mean that the protocol should have to, at a higher level, deal with key initialization, padding sizes or what have you. The protocol should be able to parameterize this to a runtime: define encryption to use some named cipher, and *e.g.* if it is a block cipher, the runtime will provide the IV, insert necessary padding, and so on. The argument that knowing exactly how this is done is paramount for the security actualizations and thus have to be exposed to the programmer only holds if, and only if, the runtime cannot be trusted to both perform the right operations, detect them when conforming, and detect nonconforming behaviour. Exactly what to do, *e.g.* CBC-mode vs. OFB-mode, is a policy decision.

Also, for a system it is essential to determine how a public key is to be verified and validated, but this is definitely a policy issue unrelated to the protocol itself. Hence, there is no support for this in Obol.

Another issue that has become irrelevant is randomness. It seems that all modern processors and operating systems have ample supply of entropy available to programs. Hence, protocols should be expressed and implemented on the assumption that cryptographically nonces and (fresh) keys can be generated at will, and instead we can assume that the runtime will raise an exception (or equivalent) when it can determine that this service is not available.

Naming. All non-trivial protocols assume the existence of a naming scheme, especially in relation to public keys where it must be determined which key to use for a particular purpose.

Naming is inherently a binding to a local identity. By this we mean that a subject is named, and the resulting binding is to some idea of identity that only exists locally where the binding is made. This is even true for SPKI's naming scheme, which allows local names to be used globally [19]. The actual identifying binding only exists when and were it is evaluated, and thus locally.

Exactly what this naming scheme consist of, is unclear in the general case. For instance, a message such as

$$\textit{Message } n \quad A \rightarrow B : A, B, \{A, B, \ldots\}_{K_{ab}}$$

assumes that A, B inside the message, and the encrypted part, somehow identifies the participants A and B. Can A and B be TCP/IP endpoints, or are they named keys in a X509 hierarchy, or MD5-sums of public key certificates, or maybe complete public key certificates? Also, K_{ab} is somehow associated with A and B, so there are actually four ways which the names A and B are used, but it is not at all clear how they are used.

This would indicate that names can have different type, and also different representation. If such representation have security implications for the protocol, then these should be visible when analysing the protocol.

We do not know how to deal with naming in a consistent manner. In particular, it is not clear whether naming is "just" an safety measure or if it has other security implications In fact, we strongly suspect that it is unwise, security wise, not to make the cost associated these naming issues explicitly visible. In the traditional notation names can be sprinkled throughout the protocol [20]. To our knowledge, none of the analysis tools concerns themselves with the representation of names, other than assuming it is possible to generate and parse names, and bind them to an identity: names are treated like symbols, and used only for identification.

One possibility is to use a naming system that facilitates just the binding of local names to global ones; SPKI has been designed with this in mind [21]. However, an implementation of SPKI is itself a non-trivial issue [22].

Naming is one of the issues Obol has been designed to investigate. The crux of the matter is that in the above example, the name "A" is used in four different ways, while it is not at all clear how. We wish to use Obol as our research vehicle to examine this.

Representation of Messages. If the representation of names have no implications for a protocol's security properties, then the representation of any message part have nothing to say for the protocol's security properties. Matters concerning the representation of messages, or message components, are therefore best dealt with by a protocol runtime environment.

Again, this is not an argument for abandoning sound engineering. Our argument that message representation have no consequence for a security protocol's security properties only holds if a protocol runtime is able to "export" security properties, *i.e.* giving guarantees such as ensuring that padding is correctly applied and verified before removal, or that names can be compared. We believe that such a distinction actually enhances the impact of a security protocol, since we now very clearly can specify the requirements under which it will work.

Order of Message Components. An issue which is not at all clear, is whether the order of components in a message has security implications or not. If we for

a moment discard the engineering fact that two peers with different assumption on ordering will be unable to communicate, the core of the issue is: Does the order have security implications?

If the answer is affirmative, then, if nothing else, we note that there must be problems applying BAN to the analysis of this protocol. As we know, in BAN the order of components is explicitly not important. On the other hand, if the ordering has no security implications, we can leave it to the runtime to deal with it.

Like naming, ordering of message components is an issue which we will use Obol to examine further.

2.3 Summary

As functional requirements we will demand explicit sending and receiving of messages, encryption primitives, and availability of a high-level naming scheme. There is no need for low-level formatting, padding and other pure engineering aspects. Basically we have used the same line of argument as is used by BAN. The success of BAN demonstrates beyond doubt that important properties of cryptographic protocols are captured by the symbols of the logic, and the rules of interference capture what they mean. The challenge is then to bring the high level abstractions into the real world in a way that is as faithful as possible to both the abstractions, and their realizations.

3 Design of Obol

Having discussed the functional- and non functional requirements of Obol, and our choice of using the world view offered by BAN, we will now discuss the trade-offs when we want to design a language on these terms.

3.1 Layers of Abstraction

Because we want to keep the level of abstraction as high as possible, that is, as close to the language dealing with security properties of a protocol, we cannot concern ourselves with lower-level details not relevant to the security issues.

To make these low-level details reachable, we draw inspiration from the field of middelware, and segregate the different levels of abstraction into components accessible through a well-defined interface. In particular, we want to use the ideas of reflection and composition to achieve a runtime supporting pluggable low-level functionality. This means that we treat message representation and transformations up and down various layers of abstraction as pluggable components. See Figure 1.

The goal is to use the same model of interaction between high-level language constructs and low-level functions, between protocols at the high level of abstractions, so that protocols can use other protocols, as well as using this model for accessing the runtime functionality from an application's point of view.

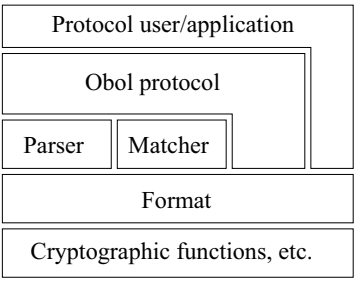

Fig. 1. Architecture. The Format layer deals with message representation, while the Parser and Matcher layers deals with message composition and recognition. In implementations, some overlap of these three layers will occur.

3.2 Syntax and Interpretation

We want the syntax of a protocol language to simple enough so that writing a parser should be straight forward, with as few terminals in the language as possible. To that end, we borrowed ideas from Lisp, so we have a functional prefix Lisp-like syntax, where program and data share the same structure, and where symbols are placeholders for properties and values, which have type. We investigate some of the language's primitives below.

A program written in Obol consists of a list of statements. Statements returns values that are implicitly assigned to the context in which the statement is executed. The language is not purely functional, however, as there are calls that are called solely for their side effects.

Obol is not a logic, and beliefs need not remain true as they do in a modal logic. In particular, beliefs need not be stable [15].

3.3 Language Constructs

An implementation of a protocol has to deal with a given participant's point of view, *i.e.* it only describes one side of the protocol. This means that for each participant in a protocol there exist a local view of the protocol state, and every statement regarding other participant's view of the protocol state will be guesswork. The design of the protocol as a whole serves to make this guess as good as possible. However, an implementation of a particular participant's rôle in the protocol cannot allow itself to speculate on the validity of remote state, and can only consider it's own state with certainty. This local state can only be updated by a participant's explicit actions, that is, what it contributes by itself, or what it accepts from "outside." In reality, only the receipt of a message warrants a state change as sending a message does not imply it will be received. The primitives of a protocol language must reflect this.

There are constructs not discussed here, including operands for loading and saving data to files, returning protocol results, attaching protocols to representation-level components, looping, error and exception handling. We believe none of them have security implications (outside the realm of engineering).

Local State. There are two primitive operators that changes the local state of a protocol implementation end-point: `believe` and `generate`.

Our world view is that which is entrenched in BAN. The principal *believes* something about his own state, and that of his peers. The `believe` operator updates local state updated with some given information about some symbol. We allow symbols to have multiple values of multiple types, which allows us to deal with complex symbols such as names, where *e.g.* A can designate both a TCP-connection and a textual name.

The `generate` operator updates the local state by constructs new data of a specified type. Because the runtime is trusted to provide data of high enough quality, generated data is also implicitly believed to be good:

```
(generate R shared-key AES 128)
```

Here a 128-bit AES-key is generated and assigned to a variable named R.

The programmer can change what is believed about a given symbol:

```
(generate Q nonce 128)
(believe K Q ((type shared-key)(alg AES)))
```

Here a 128-bit nonce is generated and assigned to symbol Q. The content of Q is then explicitly believed to be good as an encryption key and then stored in the variable K. The reason for storing the information with two types might be that it needed both as a nonce and as a key.[1]

Having access to a key does not imply that any particular beliefs are held about the key, except that the bits indeed do represent a key. In BAN, no distinction is made between having a key and having belief in it; this is altered in later efforts along the same lines [23]. Because Obol is a programming language, it is the programmer that decides the beliefs he deems it justified to hold on a key. The language only enforces that beliefs on the type must be made explicit.

Sending and Receiving Messages. The `send` primitive constructs a message from a list of symbols (or the primitives `encrypt` and `sign`) and sends it to a known protocol participant. It is left to the runtime to implement an encoding of the components that makes parsing and reconstruction possible. No assumptions are made on the properties of the communication primitives used to implement message transmission. If, for example, ordering of messages is important in the protocol (which is almost always the case) elements must be added (by the runtime) to the messages to ensure correct ordering.

The only way of being influenced by other participants in a protocol, is to receive messages from them. The `receive` primitive takes a specification (a *message template*) of what the expected message is supposed to contain, using symbols and primitive operations that are the converse of those used by the `send` primitive. In addition to recognizing known message components, we "recognize" previously unknown ones by assigning them to anonymous variables (symbols

[1] Which is suspicious from a security point of view, but here we are just demonstrating the power of the language.

prefixed with an asterisk, *e.g.* "`*A`"). The anonymous variable can later be given type using the `believe` primitive, or it could just be kept as an opaque binary data object.

In the following protocol, a nonce is generated and sent. A reply is then expected containing the same nonce, and a peer-provided new one.

```
(believe A "somehost:1234" host)
(generate Na nonce 128)
(send A Na)
(receive A Na *Nb)
```

The call to `receive` blocks, and returns when a message containing both the nonce and a unknown component is received. The message template is here only a nonce and an anonymous variable; more complex patterns are possible when dealing with encrypted material (see below). The programmer gives the system a hint on where to look for the message, instead of having the runtime inspecting all incoming communication, which can have severe performance implications if the inspection involves decryption or verification. If the programmer don't know or care where a message would come from, an anonymous variable could be used instead, assuming the runtime supports arbitrary reception of messages (which can be the case in *e.g.* a middelware infrastructure).

Following our design of considering only one side of the communication, receiving a message is a local event. As such, the receipt of a message is not detectable by others. This view of message passing is in accordance with the model of computation found in [23, Sec. 5].

Cryptographic Operators. There are two classes of cryptographic operators, for message transformation, and for verification. Message transformation is done with the `encrypt` and `decrypt` operators. They are similar to `send` and `receive`, but the specified message is sent to/received from a key.

The `encrypt` operator takes a key and a message specification, and yields a binary ciphertext message component. Encryption is parameterized by the key, so believing something about the key changes the result of using it (*e.g.* changing the associated cipher algorithm).

Like `receive`, the `decrypt` operator accepts a message template for what is expected to be "received" by decryption, as well as assignment of unknown message components. The input ciphertext is either provided by the environment (line 1), or explicitly stated by the programmer (line 2):

```
1. (receive A (decrypt k "foo" *F4))
2. (decrypt (k (encrypt k "foo" 65537)) *1 65537)
```

Message verification is supported by the operands `sign` and `verify`, which accepts a key and a message specification, and either produces or checks a signature value.

```
(verify (Kpublic (sign private "foo" 65537)) "foo" 65537)
```

Like `receive`, `verify` can obtain the signature value to check either explicitly or implicitly from the context. Verification failure outside the context of `receive` is an error condition, while inside it means that the message wasn't received.

In all signature schemes we are aware of, one must explicitly specify what the signature is on. Also, the signature can be on data that is not actually sent, but computed. We believe that the `sign` and `verify` operators captures this explicitness. Even if the trade-off is repetition of message specifications, we cannot see a better (*i.e.* more elegant and general) way of doing this.

4 Implementation

4.1 Prototypes

We have several implementations of our ideas, although none are complete.

We started using Common Lisp to do a rapid prototype – this has also obviously influenced Obol syntax, and we got inspired by the dynamic programming environment.

Since one of the main application domains seems to be middelware (see Section 5.1), effort was invested into applying Obol in one of the more experimental reflective middelware platform: OOPP[24].

To test our hypothesis that the Lisp-inspired roots of Obol didn't influence our ideas, an imperative-style version was implemented.

The latest version of Obol returned to the original "Lisp-ish" notation, but implemented the runtime in Java, the goal being integration in an industrial middelware platform (JBOSS).

We use the name "Lobo" for a given implementation of the Obol runtime.

4.2 Prototype in Java

Implementing an Obol runtime in Java allows us to demonstrate applicability, as well as explore how to best utilize Obol/Lobo from a protocol user (*i.e.* application) standpoint. Our current approach is to instrument Lobo using the Java Management Extension (JMX), and is ongoing work. A schematic over the current design structure can be found in Figure 2.

The default message representation is the Java Serialization format. An example test-run from the Java environment using the Serialization format is shown in Figure 3. Although Serialization allows for all kinds of interesting behaviour, such as sending classes and other complex objects, it would appear that most security protocol messages consists of primitive data types, *e.g.* integers, strings and byte-arrays. Since the default transport is Serialization, we believe it will serve as a useful measure for how difficult it will be to provide other representation formats. If a protocol requires a format as powerful as Java Serialization, then it would endanger our hypothesis that a protocol's representation format is irrelevant. That said, we do see possibility of constructing a protocol that only works using Java Serialization, but we argue that in such a case one is actually

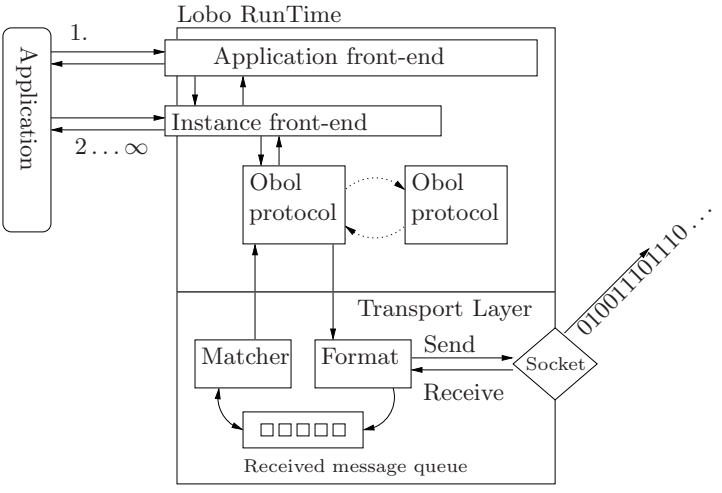

Fig. 2. Java Lobo implementation structure. The application first contacts the application front-end to obtain a binding to a protocol instance, and interacts with this instance from then on.

```
% (generate k shared-key AES 128)
## "k" = "javax.crypto.spec.SecretKeySpec@17720",
    (("generation spec" "shared-key AES 128")
     ("defined by" "generate")("defined line-number" "1"))
% (believe *a (encrypt k "foo" "bar" 65537))
## "*a" =
     0: 165C 5484 41B1 0F3F AB9D E834 3D63 1E1F   ..T.A.....4.c...
    16: A1E6 4267 D5F8 2831 D03B 6D1F 5739 45B2   .¾Bgí¿.1?.m.W9E.
    32: 45C9 9C36 AFF8 08F3 E1CF 3DF2 9194 D306   E.6.¿.ì.Ëê......
    64: 19BB 05AA 289D 0065 2EF0 F70B 8902 B087   .......e.?......,
    (("type" "binary")("defined by" "believe")
     ("defined line-number" "2"))
% (decrypt (k *a) *1 "bar" *3)
% (believe *3 ((type number)))
## "*3" = "65537",
    (("changed at line" "4")("type" "number")
     ("changed by" "believe")("defined by" "decrypt")
     ("defined line-number" "3")("number-of-bits" "32"))
```

Fig. 3. Verbose debugging output from a test-run of Java implementation of the Obol runtime

focusing on achieving compatibility with an existing application, which is not the focus of our efforts.

The Java prototype is currently the most active Obol project.

4.3 Examples

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

Implementation of A:

```
1 [self "host-A:9000"]
2 (believe B "host-B:9000" host)
3 (believe Kas (load A)
     shared-key ((alg AES)))
4 (generate Na nonce 128)
  ;; Message 1
5 (send B A Na)
  ;; Message 3
6 (receive *S (decrypt Kas
      B *Kab Na *Nb) *toB)
7 (believe Kab *Kab
     ((type shared-key)(alg AES)))
  ;; Message 4
8 (send B *toB (encrypt Kab *Nb))
```

Implementation of B:

```
1 [self "host-B:9000"]
2 (believe S "host-S:9000" host)
3 (believe Kbs (load B)
     shared-key ((alg AES)))
  ;; Message 1
4 (receive *A *Na)
5 (generate Nb nonce 128)
  ;; Message 2
6 (send S B (encrypt Kbs *A *Na Nb))
  ;; Message 4
7 (receive (decrypt Kbs *A *Kab) *2)
8 (believe Kab *Kab
     ((type shared-key)(alg AES)))
9 (decrypt (Kab *2) Nb)
```

Implementation of S:

```
 1 [self "host-S:9000" host]
   ;; Message 2
 2 (receive *B *fromB)
 3 (believe Kbs (load *B) shared-key ((alg AES)))
 4 (decrypt (Kbs *fromB) *A *Na *Nb)
 5 (believe *A ((type host)))
 6 (believe *Na ((type nonce)))
 7 (believe *Nb ((type nonce)))
 8 (believe Kas (load *A) shared-key ((alg AES)))
 9 (generate Kab shared-key AES 128)
   ;; Message 3
10 (send *A (encrypt Kas  *B Kab *Na *Nb) (encrypt Kbs  *A Kab))
```

Fig. 4. An Obol implementation of the Yahalom protocol

Yahalom. The Yahalom protocol (taken from [25, page 30]) has four messages as follows:

$$\begin{array}{ll} \textit{Message 1} & A \rightarrow B : A, N_a \\ \textit{Message 2} & B \rightarrow S : B, \{A, N_a, N_b\}_{K_{bs}} \\ \textit{Message 3} & S \rightarrow A : \{B, K_{ab}, N_a, N_b\}_{K_{as}}, \{A, K_{ab}\}_{K_{bs}} \\ \textit{Message 4} & A \rightarrow B : \{A, K_{ab}\}_{K_{bs}}, \{N_b\}_{K_{ab}} \end{array}$$

Figure 4 shows an Obol implementation of this protocol; the line numbers are not part of the implementation, and the numbers in comments (non-numbered lines starting with "`;;`") refer to the protocol description above.

Notice in the implementation of S that we have chosen to promote the two components into nonces (lines 6 and 7). This is not strictly necessary, as they could have been included in the encryption expression in the line 10 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.

Implementation of A:

```
1 [self "host-A:9000"]
2 (believe B "host-B:9000" host)
3 (believe P (load "P.key")
      shared-key ((alg AES)))
4 (generate Kt keypair RSA 512)
5 (send B A (encrypt P
    (public-key Kt)))
6 (receive (decrypt P
    (decrypt (private-key Kt) *Kab)))
7 (believe Kab *Kab
    ((type shared-key)(alg AES)))
```

Implementation of B:

```
1 [self "host-B:9000"]
2 (believe P (load "P.key")
     shared-key ((alg AES)))
3 (receive *A (decrypt P *Kt))
4 (believe *Kt ((type public-key)
               ((alg RSA))))
5 (generate Kab shared-key AES 128)
6 (send *A (encrypt P
             (encrypt Kt Kab)))
```

Fig. 5. An Obol implementation of the two first messages in the EKE protocol [26]

EKE. Eke (Encrypted Key Exchange) was proposed by Bellovin and Merritt [26] 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:

$$\begin{aligned} &\textit{Message 1} \quad A \rightarrow B : A, \{K_t\}_P \\ &\textit{Message 2} \quad B \rightarrow A : \{\{K_{ab}\}_{K_t}\}_P \end{aligned}$$

The Obol implementation is shown in Figure 5.

4.4 Conclusion

The implementation of Obol is fully functional, is written in 100% pure Java, and can be embedded in applications and systems. The latest version can be obtained by contacting the authors. The small examples shown here gives a mere flavor of the language. We believe that almost all of the protocols in the Clark-Jacob library[2] can be implemented in Obol.

5 Application

As Obol makes it possible to write and deploy protocols it is prudent to ask where it is reasonable to apply them. Because Obol cleanly decouples protocol implementation from protocol execution we can apply heavy-weight protocols in settings where they normally are outside reach. We have identified three challenging settings:

[2] An updated version of this collection is currently available at `http://www.lsv.ens-cachan.fr/spore/`

1. Providing configurable protocols to components in a middleware system;
2. Indirectly executing heavy weight protocols on smartcards;
3. Revocable protocols; infrastructure to ensure that a service provider can revoke a protocol.

Below we will elaborate on these applications of Obol.

5.1 Middleware

General middleware environments offer an execution environment for components, usually called a capsule or a container. Components implement the business logic of applications, while the container provides logging, communication facilities, and so on. This computational model works well when the (set of) components together with the container, are self-contained. The situation becomes murky when components need to rely on external services, in particular when these services must be accessed in a secure manner. Such services might be to convey funds, make reservations, and so on. The problem is the inherent complexity in security protocols, be they advanced authentication technology or simple encryption, where keys must be obtained from key distribution (*e.g.* certificate) infrastructures. Regardless of purpose or origin, code necessary to conform to a particular cryptographic regime must be implemented on the client side. The problem is then how to easily adapt components to changes in the interfaces offered by services, and still avoid that all components must carry the code to execute all these protocols. For example, if a server changes from SSH-1 to SSH-2 or from SSL to TLS, this should be a minor change; it does not at all change the business logic. However, if a new implementation must be provided at the client side, it is a major obstacle. There are two possibilities: Either change the deployment descriptor of the component and redeploy, or rely on the (provider of the) container to provide the code. Since both approaches have obvious disadvantages, the net effect is a striking lack of flexibility, very much in contrast to the original goals that made this programming model attractive in the first place.

By using Obol, components can implement cryptographic protocols without having to rely on particular protocol implementations in the container. Also, because Lobo would become the focal point of a variety of protocols, analysis of their implementation becomes feasible.

Components confined to a container communicate with external services by means of Lobo. Lobo might itself be implemented as a component, and provide an interface where other components (called clients) download programs (protocols) to be executed by the coprocessor. Other implementations are also possible, *e.g.* LOBO could be part of the container implementation, or be offered as an external service accessible through the container.

One of the main problems is that we need a secure channel to LOBO. A component running in a container will have to trust the container to provide this channel, in effect incorporating the container into the component's trusted computing base (TCB [5]), a scenario which is true in any case. If LOBO is located outside the immediate environment, a secure channel has to be established first using some technique for authentication and privacy.

5.2 Smartcards and PDA

Another setting where Obol can be applied is systems that use equipment with limited processing, storage, and communication facilities, such as smartcards. Here the problem stem from the challenge of reprogramming a large number of cards, but also from the fact that the resources available in smartcards are scarce. Obviously this holds for all kinds of devices deprived of resources.

In the middleware setting, communication between LOBO and the component is provided by the container. In a setting with smartcards, a shared key can be installed in the card and used to establish a secure channel to a server (which is part of the TCB). The shared key, when trusted by both sides, represents a trusted channel that provides both authentication and privacy. The program, written in Obol, is then transferred from the card to the server, the server executes the program, and sends the results back over the channel realized by the shared key. Knowing how scarce resources are in smartcards, using LOBO and Obol might make new and exciting applications possible [27].

Obviously, applying Obol and Lobo to leverage a server does not solve the fundamental problem that a weak machine isn't able to perform all the tasks the user would like. In particular, the TCB increases which is a problem not to be viewed favorably. On the other hand, delegating authority (over protocol execution) is an extension of the TCB that is not so hard to analyze.

The great advantage of smartcard is their tamper-resistance, and that this makes it possible to build a TCB that is distributed. In most cases, physical access to the computing device makes it impossible to trust its integrity, but this is normally not the case with smartcards. This, however, is countered by their lack of flexibility. They are hard to program, and the trust model surrounding their use is complex. In particular, smartcards without a channel on which feedback can be offered to users are prone to a wide range of attacks that are hard to defend against [28,29].

By using Obol, applications on the card can engage in more complex protocols without having to have the full implementation on board.

5.3 Certificates

A third setting is one where a service can use existing infrastructure for authentication, that is, use *e.g.* an X.509 or SPKI-based PKI to distribute certificates. The server can embed in the certificates an Obol program that, when executed, realizes the client side of the protocol on which the service is provided. The client would then obtain the certificates, verify the correctness, and immediately have access to an authenticated version of the protocol needed to access the service.

In addition to allowing us to efficiently distribute a protocol, this approach also makes it possible to revoke protocols when security considerations makes this desirable.

Having revocable protocols as part of certificates is novel, and demonstrates the effectiveness of Obol as a research vehicle.

5.4 Conclusions

The overall goal of Obol is to facilitate an experimental platform for protocol research. The three, very different, areas of applications demonstrates fully that Obol can be applied in a variety of settings.

6 Related Work

The effort of Obol draws on projects from four different categories: logics, formal description techniques, protocol implementation languages, and middleware solutions.

6.1 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 [30]. 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 [31]. There are several classes of tools available for such analyses, ranging from modal logics [12,32,33], to Higher Order Logics rewrite systems [34], and to state attainability deducers [30,31]. 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.

Obol is definitely not a logic, but rather a high-level programming language. However, the abstractions the language offer has been chosen to make the transition from an analyzed protocol to implementation as simple as possible. The relations to BAN are obvious, and the *raison d'êntre* for Obol is, after all, to give those working at a higher level of abstraction the means to implement their protocols and being reasonable confident that the implementation adheres to the protocol description used in the analysis.

6.2 Formal Description Techniques

LOTOS (Language Of Temporal Ordering Specification) was developed by OSI in the late eighties as a Formal Description Technique [35,36]. It is a language for the description of protocols. It has complementary formalisms for 'data' based on ACT ONE [37] and 'control' based on CSP [38]. Estrel is one of the family of languages used to describe real-time systems using the state machine model [39,40].

This approach is fundamentally different from the one taken in Obol. A protocol implemented in Obol is not a vehicle for analysis but rather a language as close to the original specification as possible.

Another approach is to assume that TCP/IP is available, and then place a new layer between the application and the transport layer [41]. 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.

Obol is related also to CAPSL, which is a high-level language for applying formal methods to the security analysis of cryptographic protocols. Its goal is to permit a protocol to be specified once in a form that is usable as an interface to any type of analysis tool or technique, given appropriate translation software [42]. Protocols specified in CAPSL are translated into an intermediate format (named CIL), and from there to any format needed by the tool that is to be used. Related to Obol, there are tools that from CIL generates Java [43] and Athena [44,45]. CAPSL features a clearly defined notion of types, encryption and the sending and reception of messages. This project has extended the effort also to analyse secure multicast protocols [46].

Although related, Obol differs substantially from CAPSL in that the latter is designed to assist in the analysis of protocols, rather than be executed "as is."

6.3 Implementation Languages

Prolac is a language for implementation of protocols [47]. The language can be compiled into C, and, as is the case with Obol, both sides need not be implemented in the language. Prolac compiles to C and the idea is that the resulting code can be embedded in an application or system. Because the language has been designed to facilitate the implementation of "TCP-like" protocols, Prolac has support for low-level issues such as buffer management, and the inclusion of code written in C. Both of these would thwart the efforts of Obol to ensure that such low-level issues do not pollute the protocols.

A very different approach is to start out with cryptographic primitives and generate a protocol from the elements used in the encryption [48]. The resulting protocol, which is to be embedded in both sides of the communication, is claimed to be provable secure because it is derived from a mathematical problem witch is provable hard. Such an approach is very limited compared to Obol, and there will be no flexibility on how the protocol is to act.

The design of Obol has been guided by our desire to capture in a programming language powerful abstractions from the domain of security protocols. The language Morpheus was designed in precisely the same manner, but for a different setting [13]. Also this project aims for the implementation of "TCP-like" protocols, and as such only the methodology of the project is directly relevant for us. An interesting observation regarding Morpheus is that the authors claims that Morpheus is situated between the two extremes represented by Formal Description Techniques (see above) on one end, and implementations in a general-purpose language constrained only by the operating system on the other [13, Section II] This is exactly the same position as we claim for Obol, but in the security protocol domain. Morpheus was not fully implemented.

6.4 Middleware

Middleware solutions that are related to Obol can be divided in two: Systems that composes protocols to achieve the sough-for properties, and systems that alter encryption properties to achieve the same.

Protocol Composition Systems. One can reasonable view Obol as middleware: A software layer that offers services to applications. In the case of Obol the service is mechanisms to construct (security) protocols. Because Obol is concerned with the construction and execution of protocols it is related to systems where an application can use underlying mechanisms to construct protocols, for example Ensemble [49] (there are others *e.g.* Antigone [50]).

However, the focus is on security *properties* that can be achieved and maintained within group communication by applying protocols, rather than how to construct these protocols. The target of Obol can be viewed as more focused, in that we are concerned with the implementation of protocols rather than their composition.

Interceptors. Both in .NET and in EJB anyone (with sufficient privilege) can alter the protocol by inserting in the stream of data so called *interceptors*. In some senses this is protocol programming. The mechanisms are designed to alter (for example by encrypting) a stream of data passing from one peer to another, and not to change the protocol all together.

"Da Capo++" is a middleware system where many of the application's needs and communication demands can be specified in terms of QoS values [51]. 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.

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.

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. First, 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. Second, 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++.

6.5 Jini

Jini is a network technology supplied by Sun as part of the Java effort. It provides infrastructure components and programming models for allowing services to be discovered by clients, and providing these clients with the necessary executable code in order to use the services.

Albeit superficially similar to Obol, in that code is made available to clients, the important distinction is that Jini is essentially a service-"driver" distribution facility, heavily relying on a Java infrastructure for code (classes) distribution, reflection and usage. It does not concern itself with protocols per se, other than possibly providing a proxy class for accessing a service over a particular protocol. It can also be argued that Jini provides implementations for both sides of the protocols it provides.

7 Discussion

The abstractions offered by Obol has been chosen so that the functionality of a majority of security protocols can be implemented. At the same time, Obol avoids making any global assumptions, and all security related issues are left to the programmer; we believe this is sound engineering. Part of this is to ensure that the protocol programmer does not control the ordering of components of messages, and thus does not rely a particular ordering for security.

Obol exposes in full the intricate issue of naming; an Obol program has embedded in it the naming scheme of the system, which we believe is sound engineering. So called SPKI names is but one possibility.

The even increasing diversity in computing milieus has led us to design Obol as a language in it's own right rather than as a set of libraries (classes) for a particular language, middleware platform or operating system.

As Obol is underpinned by an implementation this makes it possible experiment with protocol design. Furthermore, as all packages from all instances of protocols passes through the same machinery, monitoring for intrusion detection and replay attacks can be put in place.

References

1. Anderson, R., Needham, R.: Programming satan's computer. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 426–440. Springer, Heidelberg (1996)
2. Barak, B., Halevi, S., Herzberg, A., Naor, D.: Clock synchronization with faults and recoveries (extended abstract). In: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, pp. 133–142. ACM Press, New York (2000)
3. Zhou, L., Schneider, F.B., Renesse, R. V.: Coca: A secure distributed online certification authority. ACM Trans. Comput. Syst. 20(4), 329–368 (2002)
4. Department of Defense: DoD 5200.28-STD: Department of defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC), The Orange Book (1985)

5. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distribued systems: theory and practice. ACM Transactions on Computer Systems 10(4), 265–310 (1992)
6. Thompson, K.: Reflections on trusting trust. Communications of the ACM 27(8), 761–763 (1984); Also appears in ACM Turing Award Lectures: The First Twenty Years 1965-1985. ACM press, New York (1987), and Computers Under Attack: Intruders, Worms, and Viruses. ACM press, New York (1990)
7. Simmons, G.J.: Cryptanalysis and protocol failures. Communications of the ACM 37(11), 56–65 (1994)
8. Stubblefield, A., Ioannidis, J., Rubin, A.D.: A key recovery attack on the 802.11b wired equivalent privacy protocol (wep). ACM Transactions of Information Systems Security 7(2), 319–332 (2004)
9. Abadi, M., Needham, R.: Prudent engineering practice for cryptographic protocols. IEEE Transactions on Software Engineering 22(1), 6–15 (1996); A preliminery version appeared in the Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy (1994)
10. Harris, J., Henderson, D.: A better mythology for system design. In: ACM Conference on Human Factors in Computing Systems, pp. 88–95 (1999)
11. Anderson, R., Needham, R.: Robustness principles for public key protocols. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 236–247. Springer, Heidelberg (1995)
12. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. ACM Transactions on Computer Systems 8(1), 18–36 (1990)
13. Abbott, M.B., Peterson, L.L.: A language-based approach to protocol implementation. IEEE/ACM Transactions on Networking 1(1), 4–19 (1993)
14. Durgin, N., Lincoln, P., Mitchell, J., Scedro, A.: Undevidability of bounded security protocols. In: Heintze, N., Clark, E. (eds.) Proceedings of the Workshop on Formal Methods and Security Protocols, Trento, Italy (1999)
15. Syverson, P.F.: Knowledge, belief, and semantics in the analysis of cryptographic protocols. Journal of Computer Security 1(3), 317–334 (1992)
16. Parnas, D.: Software aging. In: Proceedings of the 16th international conference on Software engineering, Sorrento, Italy, pp. 279–287 (1994)
17. Blum, J.R., Goldwasser, S.: An efficient probabilistic public-key encryption scheme which hides all partial information. In: Blakely, G.R., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 289–302. Springer, Heidelberg (1985)
18. Goldwasser, S., Micali, S.: Probabilistic encryption and hos to play mental poker. In: Proceedings of the 14th ACM Symposium on the Theory of Computing (1982)
19. Halpern, J.Y., van der Meyden, R.: A logical reconstruction of SPKI. Journal of Computer Security 11(4), 581–613 (2004)
20. Abadi, M., Needham, R.: Prudent engineering practice for cryptographic protocols. IEEE Transactions on Software Engineering 22(1), 6–15 (1996)
21. Halpern, J.Y., van der Meyden, R.: A logic for SDSI's linked local name spaces. In: PCSFW: Proceedings of The 12th Computer Security Foundations Workshop. IEEE Computer Society Press, Los Alamitos (1999)
22. Myrvang, P.H.: An infrastructure for authentication, authorization and delegation. Cand. scient. thesis, Dept. Computer Science, University of Tromsø, Norway (2000)
23. Abadi, M., Tuttle, M.: A Semantics for a Logic of Authentication. In: Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing, pp. 201–216 (1991)

24. Andersen, A., Blair, G.S., Eliassen, F.: A reflective component-based middleware with quality of service management. In: PROMS 2000, Protocols for Multimedia Systems, Cracow, Poland (2000)
25. Burrows, M., Abadi, M., Needham, R.: A logic of authentication, from proceedings of the royal society. In: Stallings, W. (ed.) Practical Cryptography for Data Internetworks, vol. 426(1871). IEEE Computer Society Press, Los Alamitos (1989)
26. Bellovin, S., Merritt, M.: Encrypted key exchange: Password-based protocols secure against dictionary attacks. In: Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland (1992)
27. Rankl, W., Effing, W.: Smart Card Handbook, 2nd edn. John Wiley & Sons, Chichester (2000), ISBN 0-471-98875-8
28. Abadi, M., Burrows, M., Kaufman, C., Lampson, B.: Authentication and delegation with smart-cards. Science of Computer Programming 21(2), 93–113 (1993)
29. Stabell-Kulø, T., Arild, R., Myrvang, P.H.: Providing authentication to messages signed with a smart card in hostile environments. In: Proceedings from the USENIX Workshop on Smartcard Technology, pp. 93–99 (1999)
30. Meadows, C.: Formal Verification of Cryptographic Protocols: A Survey. In: Safavi-Naini, R., Pieprzyk, J.P. (eds.) ASIACRYPT 1994. LNCS, vol. 917, pp. 133–150. Springer, Heidelberg (1995)
31. Meadows, C.: The NRL Protocol Analyzer: An Overview. The Journal of Logic Programming 26(2), 113–131 (1996)
32. Gong, L., Needham, R., Yahalom, R.: Reasoning about Belief in Cryptographic Protocols. In: Proceedings of the IEEE 1990 Symposium on Security and Privacy, Oakland, California, pp. 234–248 (1990)
33. Syverson, P.F., van Oorschot, P.C.: A unified cryptographic protocol logic. CHACS Report 5540-227, Naval Research Laboratory, Washington, USA (1996); Parts of this paper appeared in preliminary form in [52] and [53]
34. Brickin, S.H.: Automatically detecting most vulnerabilities in cryptographic protocols. In: DARPA Information Survivability Conference and Exposition, Hilton Head Island, SC, USA (2000)
35. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. In: van Eijk, P.H.J., Visser, C.A., Diaz, M. (eds.) The formal description technique LOTOS, pp. 23–73. North-Holland, Amsterdam (1989)
36. ISO: Information processing systems — Open systems interconnection — Estelle — a formal description technique based on an extended state transition model (1989)
37. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Springer, Heidelberg (1985)
38. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
39. Boussinot, F., de Simone, R.: The ESTEREL language. IEEE Transactions on Software Engineering 9(79), 1293–1304 (1991)
40. Berry, G., Gonthier, G.: The ESTREL synchronous programming language: Design, semantics, implementation. Science of Computer Programming 2(19) (1992)
41. Mengual, L., Barcia, N., Jiménez, E., Menasalvas, E., Setién, J., Yágüez, J.: Automatic implementation system of security protocols based on formal description techniques. In: Corradi, A., Daneshmand, M. (eds.) Proceedings of the Seventh IEEE Symposium on Computers and Communications, pp. 355–360. IEEE Computer Society, Los Alamitos (2002)

42. Brackin, S., Meadows, C., Millen, J.: Capsl interface for the nrl protocol analyzer. In: Proceedings of the Symposium on Application - Specific Systems and Software Engineering and Technology, pp. 64–73. IEEE, Los Alamitos (1999)
43. Millen, J., Muller, F.: Cryptograpic protocol generation from capsl. SRI Techical Report SRI-CSL-07-01, Computer Science Laboratory, SRI international (2001)
44. Perrig, A., Phan, D., Song, D.X.: ACG-automatic code generation. Automatic implementation of a security protocol. Techical Report 00-1120, UC Berkeley (2000); This technical report was never issued
45. Perrig, A., Song, D.: A first step towards the automatic generation of security protocols. In: Network and Distributed System Security Symposium, NDSS 2000, pp. 73–84 (2000)
46. Millen, J., Denker, G.: Mucapsl. In: DISCEX III, DARPA Information Survivability Conference and Exposition, pp. 238–249. IEEE Computer Society, Los Alamitos (2003)
47. Kohler, E., Kaashoek, M.F., Montgomery, D.R.: A readable TCP in the Prolac protocol language. In: ACM SIGCOMM, pp. 3–13 (1999)
48. MacKenzie, P., Oprea, A., Reiter, M.K.: Automatic generation of two-party computations. In: Proceedings of the 10th ACM conference on Computer and communication security, Washington D.C., USA, pp. 210–219 (2003)
49. van Renesse, R., Birman, K.P., Maffeis, S.: Horus: A flexible group communication system. Communications of the ACM 39(4), 76–83 (1996)
50. McDaniel, P.D., Prakash, A., Honeyman, P.: Antigone: A flexible framework for secure group communication. In: Proceedings of the 8th USENIX Security Symposium, pp. 99–114 (1999)
51. Stiller, B., Class, C., Waldvogel, M., Caronni, G., Bauer, D., Plattner, B.: A flexible middleware for multimedia communication: Design implementation, and experience. IEEE JSAC: Special Issue on Middleware 17(9), 1614–1631 (1999)
52. van Oorschot, P.C.: Extending cryptographic logics of beliefs to key agreement protocols (extended abstract). In: Proceedings of the First ACM Conference on Computer and Communication Security, pp. 232–243 (1993)
53. Syverson, P.F., van Oorschot, P.C.: On unifying some cryptographic protocol logics. In: Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy, Los Alamitos, California, USA, pp. 14–28. IEEE Computer Society Press, Los Alamitos (1994)