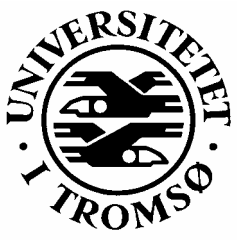

MASTER THESIS IN COMPUTER SCIENCE

The Design and Implementation of Obol

Tåle Segtnan Skogan

March 3, 2004

FACULTY OF SCIENCE
Department of Computer Science
University of Tromsø, N-9037 Tromsø

Summary

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

Obol is a high level language, and, in most cases, it is straight forward to convert a protocol description into Obol. The Obol program can then execute the protocol directly.

Obol is well suited for experimenting with security protocol design.

The Obol runtime, named Lobo, has been implemented; we present the implementation and discuss the design.

We run several well known protocols to justify the approach.

Contents

Contents 1
1 Introduction 5
Background 5
Problem Definition 5
Method 6
2 Theory 7
Domain-specific languages 7
Cryptography 7
Symmetric encryption 7
Asymmetric encryption 8
Hash functions 9
Digital Signatures 9
Security protocols 9
BAN logic 10
Prudent engineering practice for cryptographic protocols 11
SSH 12
Turing complete programming languages 13
Type systems in programming languages 13
Naming 13
Trusting the tools 14
Conclusion 14
3 Requirements 15
Obol syntax and semantics 15
Gathering and manipulating information 15
Generating data 15
Sending messages 16
Receiving messages 16
Implementing Obol 17
External server 17
Runtime 17
Trusted channel 18
Client 18
Conclusion 19
4 Design 21
Design principles for Obol 21
Obol primitives – syntax and semantics 22
Notation 23
believe 24
generate 26
send 27
receive 28
Message format 30
encrypt 32

Exceptions 32
decrypt 34
sign and verify 35
return 35
diffie-hellman 37
load and store 37
Extra operators 38
Lobo – the Obol runtime 38
Conclusion 41
5 Implementation 43
Syntax and semantics 43
believe 43
generate 44
send 44
receive 45
encrypt 45
return 45
diffie-hellman 45
load and store 45
Extra operators 46
Parsing 47
Lobo – the Obol Interpreter 47
Conclusion 50
6 Testing 51
The interactive prompt 51
Protocols 53
Wide-mouthed-frog 53
Needham-Schroeder 54
PGP 56
EKE 57
The Secure Chat application 58
Conclusion 59
7 Discussion 61
Advantages of the Obol-Lobo approach 61
Reduces the gap between design and implementation 61
High level abstractions 61
Centralizes security functionality 61
Reflection 62
Applications 62
Obol and middleware 62
Obol and smartcards 63
Obol and certificates 63
Weaknesses and future work 64
Prototype 64
SSH 64
From Obol to BAN 65

Parsing and protocol verification 66
Exception handling 66
Tight coupling with the cryptographic library 66
Binary compability with other protocol implementations 67
Performance 67
Related work 67
Formal verification 67
Implementation languages 68
Middleware 68
Conclusion 68
References 69

1 Introduction

Background

Design and implementation of security protocols to achieve security in distributed systems is difficult. Whether the goal is authentication, secrecy or non-repudiation at least three factors combine to make it hard. First, the design itself is hard: there is a long record of failed designs [22, 30], and even protocols proved to be secure by formal verification techniques fail [62]. Second, even if the design specification is based on a high level description that has been proved secure by formal verification, there may be a significant gap between the verified high level design on the one hand [60] and the unverified, detailed and complex low level design and implementation on the other hand [70]. This gap may render the formal verification worthless because security always depends on the weakest link in the chain [28, 91] and an error in the implementation will break the security. The formal verification will also be useless if it's applied incorrectly: The informal protocol description has to be transformed into a format suitable for the verification technique in question and this step is not trivial. Third, implementing security protocols involves working with low-level cryptographic primitives like ciphers, digital signatures, random numbers etc, and many programmers find that difficult [19, 32].

The Artic Beans project [7, 8] has proposed a solution to the problem of implementing security protocols: Use a high level protocol implementation language designed specifically for implementing security protocols. The language is called Obol and will contain primitives for cryptographic operations useful when programming security protocols. The main goal is to investigate if a high level description can be used directly to run a protocol in a distributed system without embarking on a separate low-level implementation effort. Therefore, the primitives will be based on abstractions which have proven useful when designing protocols. The main source of abstractions will be the traditional protocol notation found in [21], also called "BAN notation".

Obol is in an early phase and the syntax and semantics of the language is still under debate and no runtime has yet been written. It is therefore an open question whether Obol can be implemented.

Problem Definition

The goal of this thesis is to investigate the syntax and semantics of Obol:

- We will design and implement a high level language targeted at implementing security protocols. The effort will focus on finding suitable high-level abstractions which can reduce the gap between existing high level abstractions used in protocol design and low level implementations in languages like C or Java. The implementation will consist of a runtime for the new language.

Method

The report *Computing as a Discipline* [24] from the ACM describes three paradigms in computer science: Theory, Abstraction (Modeling) and Design. Theory is the mathematical approach including information theory, cryptography and graph theory. Abstraction is the modeling of problems. Design is the process of finding a solution to a problem through an iterative process by designing and implementing prototypes.

We used the design approach in this project, focusing on prototyping. Prototyping as a process acknowledges that the goal is not to produce a system, but to acquire knowledge [14]. The implementation is simply an artifact representing that knowledge. In prototyping we don't expect to get a perfect system the first time. What we do expect is a better understanding of what the system has to do. That understanding can be used in the next iteration to build a better prototype and thus leads to an iterative process. Prototyping is therefore well suited in situations where we don't know in advance what kind of knowledge we need. In this project the knowledge we sought were the proper abstractions to include in Obol. A complementary view is that prototyping as a process is a method for finding the right questions: What primitives must Obol have? What should be their syntax and semantics? The design and implementation we present in this thesis represents what we have learned and our answer to those questions.

2 Theory

Obol is based on the idea of a domain-specific language and we describe the concept. We give a brief introduction to some concepts from cryptography and security protocols, and continue with BAN logic, a common tool for analyzing such protocols. Next we introduce guidelines for protocol design and present the SSH as an example of a security protocol. We continue with two concepts from programming language theory; Turing completeness and type systems, and finish with naming and the issue of trusting the tools.

Domain-specific languages

A domain-specific language (DSL) is a programming language built for a specific application domain; rather than being general purpose it captures precisely the domain's semantics [66]. Examples include *lex* and *yacc* used for lexical analysis and parsing [35], and HTML for document markup. The idea of DSLs is old (e.g. [45]), and they offer a number of advantages over "hard-coded" program logic [76]. DLSs allow concise description of the application logic reducing the semantic distance between the problem description and the implementation [77], something we consider especially important when working with security protocols. Obol as described in [6] is a domain-specific language.

Cryptography

We briefly present some of the key building blocks when implementing a security protocol. They represent important aspects of the semantic domain for our domain-specific language.

Symmetric encryption

Symmetric encryption algorithms (also called secret key or shared key algorithms) use the same key for both encryption and decryption. When two principals communicate using symmetric encryption both need a copy of the key. This shared key must remain secret to ensure the confidentiality of the encrypted data.

Symmetric encryption algorithms come in two variants, block ciphers and stream ciphers [72]. Block ciphers break up the plaintext in constant size blocks (typically 64 or 128 bits) and give output that usually has the same length. The Advanced Encryption Standard (AES) [56] and the International Data Encryption Algorithm (IDEA) [43] are examples of block ciphers. Asymmetric ciphers like RSA can also be block ciphers. Stream ciphers operate on a stream of bits, but the distinction between block ciphers and stream ciphers blur depending on the cryptographic mode used.

The cryptographic mode determines how a sequence of plaintext blocks is handled. Electronic Codebook (ECB) simply handles each block separately. Cipher Block Chaining

(CBC) avoids weaknesses in ECB by exclusive-or'ing the plaintext block with the previous ciphertext block before encrypting. Some modes like CBC require an initialization vector (IV) to start the mode. The IV should be random data, and it must be used together with the key for correct decryption.

When using block ciphers, the plaintext will not always be a multiple of the block size. This is solved by padding the last block to the correct length before encryption. Several padding schemes exist, e.g. PKCS#5 [36], but the same scheme must be used after decryption to remove the padding.

Symmetric encryption is typically used to achieve confidentiality when transmitting data over an insecure channel, but it can also be used for authentication and message integrity in the form of message authentication codes (MACs) [38].

Key size is an important factor in the security of symmetric algorithms. 256 bits keys are considered secure [28], but 56 bits as used by the Data Encryption Standard (DES) is too small [91]. Key distribution is another security issue when using shared keys. Before any communication can take place the key must be distributed in a safe manner. Asymmetric encryption is a common way to do this.

Asymmetric encryption

Asymmetric algorithms use different keys for encryption and decryption. Public key algorithms are the most frequently used type of asymmetric algorithm. In these algorithms, the keys are always generated as matching pairs of private and public keys. The public key can be freely distributed. This solves the key distribution problem we have with symmetric algorithms; anyone who gets hold of the public key can encrypt messages that only the owner of the private key can decrypt. RSA [37] and the Digital Signature Standard (DSA) [55] are examples of public key algorithms.

As for symmetric algorithms, key size is an import factor when determining the security of a public key algorithm. Because the algorithms are based on different trapdoor one-way functions [49], the key sizes are not directly comparable, neither with other asymmetric algorithms nor with symmetric algorithms. As mentioned, public key algorithms avoid the classical key distribution problem because the public key can be freely distributed. However, that creates a new problem: How to authenticate the public key? Certificates and public key infrastructures (PKIs) try to solve that, but have inherent problems [28].

Common uses for public key algorithms are confidentiality when transferring data over an insecure channel, authentication and digital signatures. Some public key algorithms like RSA are used for most purposes while others like DSA are used for digital signatures only. Note that public key algorithms like RSA are not intended for bulk encryption, but rather for encrypting short messages like a symmetric key or message hash.

Hash functions

A hash function or message digest is a one-way function. The function takes input of arbitrary length and produces a short, fixed size output, a digest. It is one-way because for a given output it is computationally infeasible to find a corresponding input. In a hash function for cryptographic use it should also be impossible to find two messages with the same hash value. MD5 [67] and SHA-1 [57] are common hash functions.

Hash functions are often used for message integrity including MACs and digital signatures.

Digital Signatures

A digital signature is a number dependent on the message to be signed and a secret known only to the signer. The typical method is to use a public key algorithm and encrypt a hash of the message with the private key. The signature can then be verified by everyone holding the corresponding public key. RSA is often used in this way. DSA is used for digital signatures only.

Security protocols

As with any protocol in a distributed system, a security protocol is a sequence of messages between principals. The goal of security protocols, also known as cryptographic protocols, is to provide services like authentication of principals, establishing session keys, integrity, anonymity and non-repudiation, but a precise definition of the goals is not trivial [1]. Typically they make liberal use of cryptographic primitives like symmetric and asymmetric encryption, hash functions and digital signatures. The Needham-Schroeder protocol with shared keys [54] is a classic example. Other well known protocols include Kerberos [40], PGP [15] and SSH [97], and many more examples are given in [22].

The Needham-Schroeder protocol is the basis for several other protocols, for example Kerberos which again is used for authentication in Windows 2000 [50], and has received much attention. Our protocol implementation language should therefore be able to implement this protocol, and we go through the protocol in some detail to get an idea of what the language should be able to handle. The protocol is given below in the customary notation:

Message 1 A->S: A, B, Na
Message 2 S->A: {Na, B, Kab, {Kab, A}Kbs }Kas
Message 3 A->B: {Kab, A}Kbs
Message 4 B->A: {Nb}Kab
Message 5 A->B: {Nb-1}Kab

The first message tells the server S that principal A wants to communicate with principal B. A also sends a nonce Na. The server creates a new session key Kab and returns a message with nested encryption to A in step 2. A decrypts the message with an a priory shared key

between A and S. This key must have been distributed previously in some out-of-bands channel. After decryption the first term should be the nonce A sent in the first message and A must verify that this is true. A must also verify that the second message element is the name of B. The third element is the new session key. Note that A must trust S to make good keys. Implicit in this trust is the assumption that S will never reveal the key to anyone else. A can't read the last message element, but in accordance with the protocol it's simply forwarded to B in message 3, if the previous checks succeeded. When B has received message 3, A and B have shared knowledge of the key Kab and B knows that A knows the key.

The last two messages are acknowledgement messages to assure each other that the other also knows the key. When A receives message 4 she knows that B knows Kab and also that B knows that she knows the key. When B gets message 5 he knows that A knows that he knows Kab.

The kind of informal reasoning we just did for the Needham-Schroeder protocol is error prone. Because it can be difficult to even specify the goal of the protocol, it's no surprise that it can be hard to determine that the messages in the protocol achieves the goal. This can be illustrated by the discovery of errors in the Needham-Schroeder protocol soon after its publication [23]. A number of formal methods have been developed to handle such analysis in a rigorous manner [69], and the BAN logic is one of the more well known.

BAN logic

BAN [21] is a logic for analyzing authentication protocols. It assumes that authentication is a function of integrity and freshness and uses logical rules (postulates) to trace those attributes through the protocol. The analysis starts by making the initial assumptions in the protocol explicit. New assumptions are added as a consequence of receiving messages. The set of assumptions can also be extended by applying the rules to existing assumptions.

Examples of assumptions are:

- A and B believes that A and B share a secret key K.
- A sees X, meaning that someone sent a message containing X to A and A was able to understand it, possibly after decryption.

The above assumptions can be combined with the message-meaning rule to yield a new assumption (also called a belief in BAN):

> If A believes that A and B share a secret key K, and A sees X encrypted under K, then A believes that B once said K.

BAN has been successful in finding weaknesses and redundancies in protocols, but BAN also has severe limitations [48, 69].

To be able to model complex systems all models, including logics, make assumptions. If the assumptions are too powerful in the sense that they simplify the system too much, the model is no longer useful. Anderson [12] illustrates how the wrong assumptions can lead to broken security. Another problem arises when the model is transferred to a new setting where the original assumptions no longer apply. As an example, BAN has no way to model if the cryptographic primitives used in a protocol are secure and used properly. BAN simply assumes that encryption is applied properly, digital signatures verified correctly etc. This is not likely to be the case in real life. For example, message 5 in the Needham-Schroeder protocol look deceivingly simple when it's prepared for BAN analysis (idealized in the terminology of [21]):

Message 5 A->B:{Nb, A and B believes that A and B share a secret key Kab}Kab

This representation of message 5 hides a lot of complexity. A has to encrypt the nonce Nb properly using the cipher algorithm correctly (including using a cryptographic mode, possibly generating an initialization vector etc). Then A has to format the message in a way that B can understand. B must somehow be notified that a message has arrived and apply the correct key to decrypt the message and verify that the encryption worked. After decryption B must verify that Nb is indeed derived from the nonce originally created by B and sent to A in the previous message, message 4. If any of these steps are implemented incorrectly security fails. This illustrates the gap between the formal analysis done on the high level protocol definition and the low level implementation; an important motivation for Obol is to close this gap.

The assumptions in BAN has been debated at great length, see for instance [20, 85] or the overview in [86], and this reflects the importance of making the right assumptions. When Abadi *et al*. moved BAN to a new setting (smart-cards) in [2], they had to make a whole range of new assumptions and found is necessary to change the logic. Because assumptions are of vital importance for security, they should be made explicit and not hidden away. This is reflected in the design of Obol.

Prudent engineering practice for cryptographic protocols

Principles for prudent design of cryptographic protocols are set forth in [3], and also discussed in [10, 11, 28]. These principles are meant as a companion to formal verification techniques when designing a new protocol. One of the overarching principles in [3] is concerned with the content of a message: Every message should say explicitly what it means; the interpretation of the message should depend only on its content, it should not be necessary to use any context. For example, if the identity of a principal is essential to the meaning of a message, it's 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 knowledge. We believe Obol should be designed to make it easy to write protocols that follow such principles, and difficult to write protocols that don't.

SSH

SSH is a widely used protocol with many security aspects and it is natural to ask if Obol can be used to implement it. We will return to this question in the Discussion chapter, but we will describe SSH here. The SSH v2 protocol consists of three components [97]: The SSH Transport Layer Protocol provides server authentication, integrity, confidentiality and compression. The SSH User Authentication Protocol authenticates the client to the server. The SSH Connection Protocol multiplexes the secure channel into several logical channels, for instance remote shells and TCP/IP forwarding.

SSH starts by the client connecting to port 22 on the SSH server. First they run the SSH Transport Layer Protocol to establish a secure channel on this connection. Then the user authenticates himself to the server in the SSH User Authentication Protocol. Note that this protocol runs on top of the SSH Transport Layer Protocol. The client can then request one or more channels for doing remote shell, X11 forwarding etc. These channels are all multiplexed on the secure, mutually authenticated connection provided by the two previous steps.

The SSH Transport Layer Protocol starts by each side sending a string of the form ”SSH-protocolversion-softwareversion comments” [96]. Then they negotiate algorithms for encryption, integrity, signatures etc with SSH_MSG_KEXINIT messages, one message from each side. Based on the agreed upon algorithms, they do key exchange; the default algorithm for key exchange is Diffie-Hellman [26] and consists of two messages: The client starts by sending a SSH_MSG_KEXDH_INIT message containing the first part of the secret and the server responds by a SSH_MSG_KEXDH_REPLY message containing the second part and a signature. The client verifies the signature and both sides use the secret to generate keys for confidentiality and integrity (using message authentication codes). Each direction use separate keys. The two sides now have a secure channel with confidentiality, message integrity and authentication of the server. Every packet sent hereafter will be encrypted and followed by an unencrypted message authentication code (MAC). After a predetermined time (one hour) or a certain amount of data (say 1 GB), the protocol requires a rekey operation. The two sides then have to run the key exchange process again, starting with the SSH_MSG_KEXINIT message.

After running the transport protocol, the client moves to the next level by starting the SSH User Authentication Protocol [95]. The user must authenticate himself to the server, typically using passwords or public key signatures. Note that this protocol goes over the transport layer, so every message must be decrypted and the integrity verified before the messages in the SSH User Authentication Protocol can be acted upon.

If the authentication is successful, the client can continue by asking for channels. This is handled by the SSH Connection Protocol [94] and a new channel, say a shell, is started by sending a SSH_MSG_CHANNEL_OPEN request. To summarize, SSH is a complex protocol; it negotiates algorithms, does rekeying, multiplexes channels and is organized like a protocol stack.

Turing complete programming languages

A given programming language is said to be Turing-complete if it can be shown that it is computationally equivalent to a Turing machine [51]. That is, any problem that can be solved on a Turing machine using a finite amount of resources (time and tape), can be solved with the language using a finite amount of its resources.

Typically, one proves that a given language is Turing-complete by providing a recipe for translating any given Turing machine program into an equivalent program in the language in question. Alternately, one can provide a translation scheme from another language, one that has already been proven to be Turing-complete. For languages that have a lambda primitive supporting standard un-typed lambda calculus there is a shortcut; already in his seminal paper [90], Turing showed that lambda calculus is equivalent to a Turing machine in computing power.

It's important to point out the difference between being Turing complete and being a useful general purpose programming language. If that distinction isn't made, one risk stumbling into the Turing tar-pit, "a place where everything is possible, but nothing of interest is easy" [61]. We have to decide if Obol should be Turing-complete and to what degree Obol should be a general purpose programming language.

Type systems in programming languages

Types are mainly used for three reasons in programming languages [51]:

- Naming and organizing concepts: Functions and data types can be given types that reflect the way they are used in the program. This helps anyone reading the program understand how it works and why it was written a certain way.
- Making sure that bit sequences in computer memory are interpreted consistently: Type checking keeps operations from being applied to operands in incorrect ways.
- Providing information to the compiler about data manipulated by the program: For example, the type of a data structure can be used to determine the relative location of a part of this structure. This compile-time information can be used to generate efficient code for indexing into the data structure at run-time.

We have to decide how to use types in Obol and we will discuss this in the design chapter.

Naming

Naming is a central issue in distributed systems [53, 88]. A fundamental property is that a name should resolve uniquely to the resource being named [88]. For instance, when the server creates the second message in the Needham-Schroeder protocol,

Message 2 S->A: {Na, B, Kab, {Kab, A}Kbs }Kas

it relies on the ability to resolve the name B into the key Kbs shared between B and S. This adds the naming system and its implementation to the trusted computing base of any security protocol implementation that uses names in this way. In the words of Ross Anderson, naming can get “fiendishly complex” and interact with the design of secure systems in many ways [9]. We will not consider this issue further beyond stating that an industrial strength implementation of a security protocol should handle naming carefully.

Trusting the tools

Whatever you do, you must to some degree trust that your tools are working correctly. Consider Java [79]. The Jikes compiler (from IBM [34]) consists of 97.000 lines of code filling 3,7MB; we leave it as an exercise to the reader to imagine the size of the standard classes. In addition there is the VM: Kaffe has 138.000 lines of code filling 3,5MB. Add to this that you need GCC to compile Jikes. gcc-3.3 is 29MB when *compressed* and contains a whopping 1.621.051 lines of code (.c and .h) filling no less that 49MB. And you also need a working Java compiler to compile Kaffe in the first place. The average programmer has no option but to trust the tools because they are simply to large and complicated for him to verify independently. Thompson elaborated on this in his Turing Award lecture [89] and basically showed that you had to write all the tools yourself, including the compiler and operating system if you wanted to be safe.

The same argument will be relevant for us when we use Obol to implement security protocols. A certain amount of trust has to be placed in the implementation of Obol. Usually we are willing to trust the tools because the alternative is worse: When we don’t have tools (or in general abstractions), every application has to implement the functionality instead of pooling the resources into a separate tool (or abstraction) that everyone can use.

Conclusion

Implementing secure distributed systems is not possible without security protocols and accompanying cryptographic primitives like encryption. Protocol analysis is therefore mandatory and cannot be treated as some optional “add on”. But the distance between protocol analysis and implementation must not be too long; the devil is in the details and a simple implementation error will break the most secure protocol design.

3 Requirements

Obol syntax and semantics

As explained in the introduction, the goal of Obol is to close the gap between high level specifications and low level implementations of security protocols. What we need is suitable abstractions that can shield non-crypto code from protocol implementation code. The challenge is to find the correct balance: If the abstractions are too weak and low level, security related code seep into the application code. If they are too high level, applications with special needs loose the ability to influence security. The last point can be illustrated with an example from TCP/IP networks: One of the goals of IPsec is to provide authentication and confidentiality between endpoints in a network in a manner transparent to the application [38], in other words a powerful, high level abstraction. The authentication part will be able to authenticate the user in the other end of the connection in addition to authenticating the IP address he connects from. But the application is unable to get hold of this information about the user because the system calls in traditional operations systems will only return the IP address; the extra information is hidden inside the abstraction.

Gathering and manipulating information

Since assumptions and what we believe is true at a given point in the execution of a protocol is vital for security, we want a primitive for this functionality. As described for the BAN logic in the theory chapter, we may need this primitive in two situations. First, we can use it to make assumptions. Any non-trivial protocol has to make some assumptions. For instance we may load a locally stored secret key or certificate and assume they are valid and of high quality. The second situation is when we receive facts in messages, verify them and subsequently believe them to be true. Facts can be timestamps and nonces; the Kerberos protocol [40] uses timestamps and when a message is received, the timestamp is checked for freshness.

After we have gathered information in the form of assumptions or new beliefs, we may want to manipulate it or store it for later use. The primitive should therefore support this.

R1: There shall be a primitive for gathering and manipulating information.

Generating data

Many protocols need to generate new data. Data can be encryption keys, nonces and timestamps to assure the freshness of messages and so on. It seems convenient to have a separate primitive for this.

R2: There shall be a primitive for generating new data.

Sending messages

There are two challenges when sending a message: First, there must be an address so the message can be delivered to the correct receiver. Second, the message elements must be formatted in a way the receiver understands. In addition it must be possible to encrypt the message before sending it.

R3: There shall be a primitive for sending messages. The primitive must support encryption.

Receiving messages

In many protocols you have to decrypt (parts of) a message before you can accept it. Therefore the operator used to receive messages must be able to decrypt in order to verify the message. And for such a verification to be possible, it must be known not only which encryption key to use, but also what to look for after decryption. If we use the Needham-Schroeder protocol as an example:

Message 1 A->S: A, B, Na
Message 2 S->A: {Na, B, Kab, {Kab, A}Kbs }Kas
Message 3 A->B: {Kab, A}Kbs
Message 4 B->A: {Nb}Kab
Message 5 A->B: {Nb-1}Kab

We see that the reply to the first message is encrypted; an encrypted message is indistinguishable from random data. Thus, before "receiving" message 2, the message must be decrypted with Kas, and the string B be identified together with the nonce Na. The third element in the message is a random string of bits, which is to be interpreted as the shared-key Kab (with the additional understanding of which algorithm to use). The fourth element is random bits that will be forwarded to B. From this we see that receiving a message compromises the following machinery:

- Actual communication over a network. This basically means managing interfaces towards the underlying operating system.
- Parsing and interpreting the message according to a message format. Unencrypted message elements can be parsed right away. Encrypted message elements must be parsed after decryption.
- Decryption of message elements. This includes determining which algorithm to use, and to verify that the decryption Per Se succeeded.
- Matching of some elements against supplied data.

R4: There shall be a primitive for receiving messages. The primitive must support decryption of message elements and matching of message elements against data both before and after decryption.

Implementing Obol

We want a way to test the language we are designing because an implementation will give valuable feedback to the design process; in particular, it can tell us if it's possible. What constitutes an implementation is not obvious. We could implement Obol as a collection of functions. The implementation could then be used as a normal cryptographic library with the added benefit of the high level abstractions mentioned above in the requirements *R1* to *R4*. In this case, Obol would not be a language in the traditional sense, but the essence of Obol, finding high level abstractions, would remain. However, we will follow the original idea from [7] and treat Obol as a programming language. The implementation will then be a runtime with an interpreter or a compiler. The programming model for Obol and its runtime will consist of four elements as described in [7]; we discuss each element from right to left as depicted in figure 1.

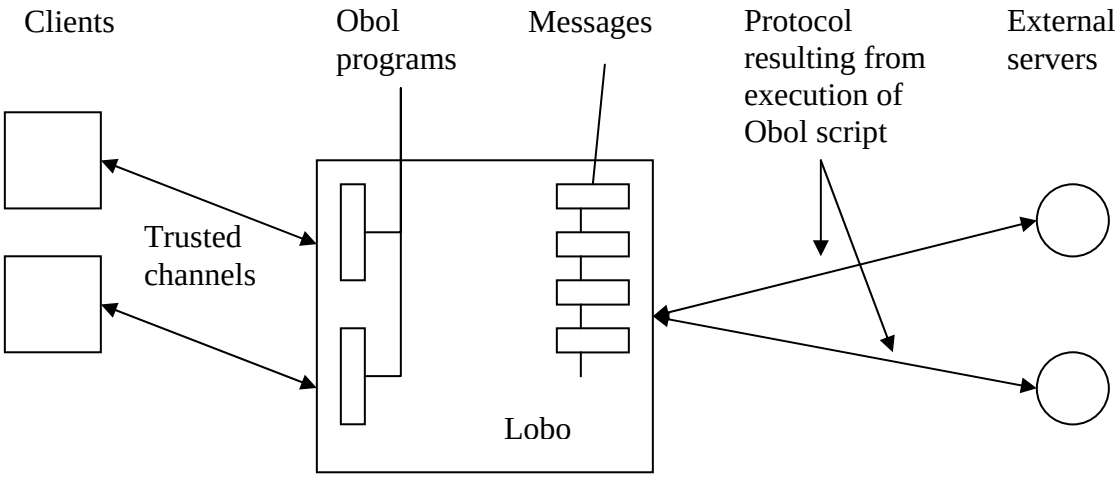

Figure 1 The programming model for Obol and its runtime, called Lobo.

External server

A typical way to use Obol will be when external servers offer services after negotiations by means of authentication protocols. Obol will be used to implement the client side of the authentication protocols.

Runtime

The job of the runtime, hereafter called Lobo, is to parse, load and execute programs[1] written in Obol. That is, Lobo communicates with the external server and executes a

[1] We will use the terms program and script interchangeably to describe a piece of Obol code.

protocol on behalf of the client, and must be able to return the result of the protocol to the client. As described in [7], Obol can be used in a middleware setting with multiple clients accessing multiple services in parallel. The runtime should therefore be able to handle multiple clients and Obol programs in parallel. Performance will not be an issue at this point in time because we want to concentrate on finding the proper abstractions and see if it's possible to implement them at all; we work in the spirit of a famous quote "premature optimization is the root of all evil in programming" [33]; or in other words, our implementation will be a prototype to explore important abstractions, not an industrial strength implementation with high performance, rigorous error handling and fully defined semantics for every special case.

Trusted channel

To use Obol, the client must download an Obol program to Lobo for execution. This requires a secure channel between Lobo and the client: Partly, this is because the runtime may want to control who is allowed to use its resources. Another reason is that the result of the protocol will typically be a secret, like a shared key, that no one but the client must know. How this is done depends on the setting. For instance, the client could use Lobo as a linked library so that the client and Lobo will run in the same address space. Then there will be no need for encryption or authentication because we have an implicit secure channel. The same applies if the client is a component running in a middleware environment and Lobo is implemented as part of the container.

Client

The client is a process that uses Lobo to communicate with an external server. The client does this by setting up a secure channel with an instance of Lobo, loading up an Obol script over the channel and waiting for the result from the runtime.

The above discussion leads to the following requirements for the runtime:

R5: The Obol runtime shall accept requests from clients for a secure channel.

R6: The runtime shall accept Obol programs over this channel and be able to parse, load and execute them.

R7: The runtime shall handle multiple programs and clients in parallel.

R8: The runtime shall make the result of the protocol available for the client.

Conclusion

Based on the understanding of security protocols from the previous chapter a language for implementing security protocols must support the following operations:

- Gather and manipulate information such as the address of a peer, trusted keys stored locally and data received in messages.
- Generate random data such as keys and nonces.
- Send messages, possibly after encryption.
- Receive messages, possibly followed by decryption.

We believe the requirements achieve this and form a sufficient basis for designing the system.

4 Design

Design principles for Obol

What properties must Obol have to be able to implement security protocols? When designing a new programming language there are many possible design choices. MacLennan [46] lists 19 design principles that can lead to very different languages depending on what principles one decide to prioritize. We want a language for implementing security protocols, nothing else, and MacLennan [46] states a simplicity design principle:

> "A language should be as simple as possible. There should be a minimum number of concepts, with simple rules for their combination."

This principle is important for us because complexity is the enemy of security: A complex language will be more difficult to learn and use correctly. It will also be difficult to implement correctly. The last point is especially important since an error in the Obol implementation will propagate to all applications using it.

We think that Obol should be a simple language, but how simple? In particular, should Obol still be a general purpose, Turing complete programming language? We think not in accordance with the simplicity principle stated above. However, there will be a price to pay in the sense that Obol will not be able to implement every protocol. We support that claim with an example: When sending or receiving messages, the message elements are often the result of non-trivial processing. For instance, one of the message elements could be a checksum computed over the other elements; in general the checksum could be an arbitrary function. But to compute an arbitrary function requires a Turing complete language, or to be realistic and avoid the Turing Tar-pit, it requires a general purpose programming language. Take bulk encryption in SSH v2.0 [97] as an example. Before encryption SSH extends the payload as follows:

> msg-length || padding-length || payload || padding

This string is encrypted and SSH appends a checksum to the encrypted string:

> ciphertext || checksum

That checksum function could be anything, and if Obol cannot produce it, we are unable to implement the protocol.

We can solve problems like the checksum function by extending the Obol runtime in an ad hoc manner with the required function. After all, when a new encryption algorithm comes along we would have to do the same thing. But for a given implementation of Obol at a given point in time, there will always be protocols that cannot be implemented. Our basic assumption is that this will not happen often because security protocols are a limited domain and our abstractions should be able to cover most protocols.

As we discussed in the previous chapter, a central design issue will be to find the proper abstractions. MacLennan [46] states abstraction as one of the central design principles for a language:

> "Avoid requiring something to be stated more than once; factor out the recurring pattern."

These "recurring patterns" must come from existing security protocols and they also represent a way to test the resulting language: Obol should be able to implement relevant existing protocols in as few lines of Obol code as possible. This calls for powerful, high level abstractions. Note that the ultimate abstraction is an implementation of an existing protocol that can be called as a function, but this is not very practical because Obol should also be able to implement future protocols and variations of existing ones. We have already given up the ability to implement every future protocol, but the primitives should be so flexible (low level) that we feel reasonably sure that we can handle most protocols. This tradeoff applies to every programming language: In assembly language everything the computer has to offer is possible, but not very practical for the human programmer so we give up freedom in exchange for higher level abstractions.

The loss of freedom that high level abstractions represent is not always a bad thing. In fact we may want to make certain things impossible. MacLennan [46] reflects this in a design principle:

> "Making errors impossible to commit is preferable to detecting them after their commission."

In our case this translates to: We want it to be easy to design good (secure) protocols and hard to write bad (insecure) protocols, and we discussed principles for secure protocol design in the theory chapter.

The final design principle we want to prioritize is about regularity. MacLennan states it as:

> "Regular rules, without exceptions, are easier to learn, use, describe and implement."

This is akin to the simplicity principle and has the same benefits; the language becomes easier to learn and use and easier to implement.

Obol primitives – syntax and semantics

The three basic parts of programming language are expressions, statements and declarations [51]. *Expressions* are syntactic elements that are evaluated to return a value. Some expressions may change the state of the machine, causing a side effect in addition to producing a value for the expression. *Statements* are commands that alter the state of a

machine in some explicit way, for example x = y + 5 will alter the state of the machine by adding 5 to the value of the variable y and storing the result in the location associated with variable x. *Declarations* are syntactic elements that introduce new identifiers, for example a new variable or a new function.

We want Obol programs to consist of a sequence off expressions which are executed sequentially. We want to use expressions and not statements, because expressions make it easy to write short, nested code as is well documented by the Lisp experience [4, 29, 39, 58, 65, 82]. The code becomes short partly because we can replace temporary variables used to hold the result of a computation with the computation (i.e. expression) itself.

Ideally, we would like to have a language consisting solely of expressions without side effects, sometimes called a *pure functional language* [51], because it can be argued that it will enhance readability and correctness [16]. However, most programmers would probably find a purely functional language awkward, and we certainly felt the need to include declarations and a certain amount of side effects. In particular, we wanted to have variables.

In accordance with the both the simplicity and regularity principle stated above, we design each Obol expression to start with an operator and continue with zero or more operands. This will make the language simpler to learn and implement because there is only one way to do things [29, 58, 65]. Such syntax is often called prefix syntax in contrast to the more common infix syntax. The advantage of having a simple, unified syntax can be illustrated by a counter example from C [31]: In C, arithmetic expressions use infix syntax; function calls use a sort of prefix syntax with the arguments suddenly delimited by commas; normal expressions are delimited by semicolons and for some reason block code is delimited by curly brackets. All of this can be expressed by single notation, namely prefix syntax with parenthesis as delimiters, as in Lisp.

To gain the full benefit of the regularity in the expressions, we enclose each expression in parenthesis. This fully parenthesized design has several advantages:

- It's easy to construct operators with a variable (indeed infinite) number of arguments because you know where the expression stops [4].
- There is no need for the complicated precedence rules found in languages like C or Java [46].
- Nesting expressions is easy; the operator and operands can themselves be expressions [4].
- Long lines no longer require any special line continuation handling and/or syntax, as for instance in Python.

Notation

Comments in Obol programs are prefixed with ";". String literals are written enclosed in double quote characters. For example,

```
"A string literal."
```

is a string consisting of 16 characters. Integer literals can be written directly in base 10 or in hex with a “0x” prefix. For example,

```
0xff55aa22
```

is an integer represented by eight hex digits. Literal byte arrays starts with “#(“ and ends with “)” and the individual bytes are written in base 10. For example,

```
#(1 0 255 255 1)
```

represents an array of 5 bytes. The implication arrow, “=>”, is used in some of the code examples and should be read as “returns”; it indicates the value returned from an expression (but is not a part of Obol). For example,

```
(generate nonce ((size 64))) =>

<ObolVariable:
    size 64
    type nonce
    value #(158 71 188 85 164 251 229 52)>
```

means that the Obol expression returns an object which can be represented by the text inside the “<” and “>” characters.

We now describe the operators in Obol. The main operators are similar to those found in [7].

believe

As described in the Requirements chapter, assumptions and what we believe is true at a given point in time of the execution of a protocol is vital for security, we therefore want a primitive for this functionality. We name the primitive **believe** after a similar feature in BAN [21]. We also described how **believe** can be used in two situations: We can use it to make assumptions, and we can use it when we receive facts in messages, verify them and subsequently believe them to be true. Since we also want to have variables in Obol, **believe** would be a natural candidate for doing declarations (assignments). This leads to the following syntax:

(believe name value type attributes): This expression creates a variable with the given name, value and type. The last operand, **attributes**, can be used to give detailed type information. For instance we could specify that a variable of type `shared-key` should be 128 bits long and be interpreted as a key for the AES algorithm [56]. For every operator we try to place the most import operands first. By this we mean that an operand to the right is more likely to be optional than an operator to the left and all operands to the left of a given

operand must be included in a given expression. For **believe** we deemed the variable name to be the most important operand because every variable needs a name.

Examples:

- `(believe K 0x1122334455 shared-key)`
- `(believe T *msg-element timestamp)`

The first example illustrates an assumption. The datum 0x1122334455 represents a hex value, which is signaled by the 0x prefix. We have a value and simply assume it's a valid shared key; this expression will store the datum as a typed value (of type `shared-key`) in the variable K as a side effect. If the variable doesn't exist, it will be created. The second example represents the situation where we establish new beliefs based upon messages we receive. Here, **believe** interprets the contents of `*msg-element` according to the semantics of `timestamp` and assign it to the variable `T`. The value of `*msg-element` will typically be data received in a message. We introduce a special kind of variable to hold such data. We call them anonymous variables and prefix them with "*". Anonymous variables are used to hold data in cases where no assumptions have yet been made on how to interpret the data. This force us to make assumptions explicit with **believe** before we use the data. We do this in accordance with the overarching principle stated in [11] which says that robust security protocols is all about explicitness. For instance, receiving a shared key can be seen as a two step process, first we receive a chunk of bits over the wire and second, we explicitly **believe** those bits to be a shared key.

Because **believe** involves interpreting a value according to the semantics of some type, there is always the possibility that this is impossible. For instance, the timestamp above could be too old, i.e. expired, or a byte array could be too short to represent a 128 bits key and so on. The following example illustrates the last case. Here we try to interpret a 40 bits value as a 128 bits AES key. Note that the attributes are given as an association list [82], which is simply a way to represent a table with parenthesis; each key value pair is surrounded by parenthesis, and all the key value pairs are surrounded by an outer pair of parenthesis. We chose a table so the attributes could be specified in any order, and it also makes it easy to add new attributes. We use the same approach for the other primitives that have an attribute operand.

Example:

- `(believe K 0x1122334455 shared-key ((alg AES) (size 128)))`

The example raises at least two general questions that apply to all the primitives: First, when we detect an error, what do we do? Should we print a warning and allow the script to continue or should we abort and return some kind of error to the application? We think the best solution is to abort because an error in Obol is a security error; this is by definition since Obol is a language for implementing security protocols. The error doesn't have to be an active attack, but we think it's prudent to be careful. The return value is a bit more difficult because the best choice may depend on the languages used to implement both Obol and the application using Obol. Therefore we'll settle for a simple string describing the error. The second question raised is; how careful should we be? What should be allowed?

Should every operator require operands of a specified type (static, also called strong, typing) or should we use dynamic typing?

We chose dynamic typing instead of static typing so variables don't have type, only values do. As explained in the theory chapter, there are three main uses for types in programming languages:

- Name and organize concepts.
- Keep operations from being applied to operands in incorrect ways.
- Help the compiler optimize the program.

Neither was relevant for us. Obol scripts will be small and use high level primitives so it is less important to help the human reader understand the structure of a program with type information. The same argument also applies to the need for primitives to declare functions (like defun in Common Lisp) and introduce local scope (like let in Common Lisp). Because the programs are small we expect the programmer to keep control of which operations can be applied to which arguments so the second use of types is not relevant either. And as for the last point, Lobo will spend most of the time waiting for network traffic or doing low level cryptographic operations which are implemented in another language. As a result, Amdahl's law [93] tells us that the performance gained by optimizing Lobo will be minimal and therefore optimization will not be a priority. Since static typing seemed to give few benefits and dynamic typing was likely to simplify the design and implementation, we chose the later.

The variables created by **believe** have script global scope. There really isn't any alternative as basic Obol offers no other scope. However, we will introduce some experimental operators including **lambda** and **let** in the Implementation chapter and they will give us local scope in addition to the script global scope.

generate

Security protocols typically need to generate new data like shared keys, public keys, nonces and timestamps. To generate keys and nonces, the primitive must interface with a low level cryptographic library and we must be able to specify what kind of key we want and maybe other attributes like key size. We therefore include an operand **attributes** as placeholder for extra information in addition to the overall type.

(generate type attributes): This expression generates a new value according to the attributes. The following example generates a 128 bits shared key to be used with the AES algorithm.

Example:

- `(generate shared-key ((alg AES) (size 128)))`

This primitive is typically used in combination with **believe** to create a new variable and fill it with a value, for example:

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

An alternative syntax would have been to let **generate** create a new variable as a side effect, for example:

- `(generate K shared-key ((alg AES) (size 128)))`

This syntax was used in [7], but we felt our solution was more in line with the functional paradigm and it would also simplify the implementation. In addition, there should be only one way to do the same thing (create a new variable), otherwise the user has to investigate if the semantics of the two ways are the same.

Valid types include `shared-key`, `public-key`, `timestamp` and `nonce`.

send

Since every protocol involves sending messages, the operator for sending data to a remote host is crucial. In the Requirements chapter we stated in Requirement 3 that the primitive for sending messages should be able to encrypt them as well. Because encryption has a central role in security protocols and because the task of encrypting a message is somewhat orthogonal to sending it, we will create a separate primitive **encrypt** for encryption and restrict the **send** primitive to handle issues like packing message elements in a well defined message format and sending the message to a receiver. With a similar argument, we separate decryption from the primitive for receiving messages and create two primitives here as well, **decrypt** and **receive**.

(send receiver data): This expression sends **data** to **receiver**. Note that **send** must use a message format the receiver understands and we will discuss this in more detail below. The following example send the value in the variable K to the remote host gaupe.cs.uit.no listening on port 9000:

Example:

- `(send “gaupe.cs.uit.no:9000” K)`

The receiver can be a string literal (like “tasko.stud.cs.uit.no:9000”) or a variable of type address. We considered making it possible to let the receiver be a list of receivers for situations where the same message should be sent to more than one, but we couldn’t come up with a realistic example in the setting of security protocols so we decided not to implement it for the sake of keeping the language simple. The receiver implicitly represents the mechanism used to deliver the message and we leave it to the runtime to find a way to deliver the message. If the receiver is a string with no port number we assume the message

is for local delivery and the string represents a local channel. This is useful for testing a protocol using a single machine.

Since a message often contains more than one message element, we should be able to replace **data** with a sequence of message elements like **(send receiver element1 element2 …)**.

Example:

```
(send "tasko.stud.cs.uit.no:9000" A B Na)
```

receive

Since every protocol involves receiving messages, the operator for receiving messages is equally crucial to the operator for sending them. Logically the messages going between two principals in a protocol represent a communication channel, regardless of how they actually are delivered [13]. And of course, the channel metaphor may be a precise description of the system, for instance when there is a direct TCP connection between the two sides. Wilkinson and Allen [93] list two fundamental operands of any primitive for receiving messages; a source and a destination for the incoming message, where the source can be interpreted as the channel. Therefore, it seems natural that our primitive should have an operand describing the channel (source) of the incoming messages. And when we look at the way security protocols are described, they assume the presence of a channel with unambiguous parsing of the messages. For example, the first message of the Needham-Schroeder protocol described in chapter 2 contains three message elements:

Message 1 A->S: A, B, Na

The protocol description simply assumes that there is a channel between A and S and that S is able to recognize the incoming bits as a message consisting of three elements where the two first should be interpreted as addresses and the third as a nonce. However, when we analyze the security of an authentication protocol like Needham-Schroeder with tools like BAN [21], the analysis does not in any way depend on the channel concept, the messages are treated as discrete units and advance the state of the protocol in discrete steps (we may want to establish a secure channel as the result of the protocol, but that is another issue). So the channel concept is not important for security. In fact, it may be an example of an abstraction that is too high level, thereby jeopardizing security: We said we wanted to make it easy to design secure protocols in Obol and difficult to design insecure ones. The principles for prudent engineering of security protocols described in chapter 2 emphasized that explicitness is important. Messages should be self contained and not dependent on context. Therefore we want to avoid protocols that rely on the existence of a channel to work. Every message should contain the protocol id, the run of the protocol and the message number in the protocol [3, 28]. As a consequence, it shouldn’t matter how the message is delivered to the runtime; whether TCP, SMTP or even SMS, the context shouldn’t matter so the message can be extracted from the delivery mechanism and put in common in queue together with every other message; later on, the runtime can go through

the queue and deliver each message to the correct script with no possibility for error. To conclude, we normally want to avoid the concept of communication channels in Obol, and instead operate only on messages. However, when testing a protocol on a single Lobo instance it can be useful to name a channel for testing purposes so we want to be able to do that as a special case.

When a script decides to wait for a message, it must specify what message it is waiting for so the runtime can look for it. This can be done by specifying the contents of some or all of the message elements. The **receive** primitive should take a pattern operand that could be used in the matching. Most of the time the messages will contain elements that we don't know about beforehand, for example encrypted data. We need a way to designate these elements in the pattern and we use anonymous variables for that. Later we may want to decrypt that element or forward it in another message. Therefore, it will be convenient to keep a handle to the message element, and we design the **receive** primitive to create a new anonymous variable in the global scope when it occurs for the first time in a pattern. The value of the new variable will be the message element and we can promote it to a typed value later with **believe** if we have to.

Based on the above, we have designed the **receive** primitive as follows:

(receive pattern attributes): This primitive causes **pattern** to be applied to all available and incoming messages until a match is found. If a channel is given as an attribute, only messages arriving on that channel will be considered during matching. If we list multiple channels, we accept an incoming message on any of those channels. The pattern can simply be a sequence of elements where each element is matched in order of occurrence against the elements in the message:

```
(receive pattern-element-1 pattern-element-2 …)
```

New anonymous variables in the pattern work as wild cards accepting any value. The message is allowed to contain more message elements than required by the pattern, but not the other way around. The expression returns a list containing the message elements.

In the examples below, lines prefixed by ";;" are Obol comments. In the first example, the expression looks for messages from the remote host "gaupe.cs.uit.no" containing four message elements. The three first elements must match the values in the three variables A, B and Na, the last element can be anything.

Examples:

```
(receive A B Na *1 ((channel "gaupe.cs.uit.no:9000")))

;; This expression accepts any message containing a least
;; two message elements.
(receive *1 *2)

;; Receive any message where the channel matches "m1".
;; This is useful for testing protocols on a single Lobo instance.
(receive ((channel "m1")))
```

```
;; Receive the first available message, regardless of origin
;; or pattern.
(receive)
```

If no message matches, **receive** should block the script waiting for new messages. If no matching message becomes available within a preset timeout period the script should abort with an error message. The timeout can be set by giving a timeout attribute. The following example sets a timeout of 40 seconds.

Example:

```
(receive ((timeout 40)))
```

There is a potential problem if several protocols are running in the same interpreter and one of the scripts do a very general receive, e.g. `(receive)`, and steels messages meant for other scripts. We could avoid this by letting every script get every message and if the message doesn't match it's ignored and the script continues to wait for a new message. This is a very robust solution in the sense that no script can steal messages destined for other scripts. However, it allows ill specified protocols which break the principles for prudent engineering in [3] to continue working. We could avoid this by using some kind of "Longest Match First" matching in the selection process, but since this is a prototype, we will assume the scripts "behave well" and not refine the matching process. Note that the "Longest Match First" scheme would not be perfect either, because a malicious script (backed by a malicious application) could always monitor the network traffic (a standard ability of an attacker when analyzing cryptographic protocols [49]), determine the protocol, run and message number and in some cases (like the first message in the Needham-Schroeder protocol) do a receive with a corresponding pattern that would steel the message from the intended receiver.

It is a strong assumption to assume the scripts "behave well", but in some situations this is less important. If the Obol runtime is used as a linked library by the application, there are no other applications to worry about. Alternatively, several applications use Obol simultaneously, and the applications must follow a protocol (i.e. use an API) to get access to the services offered by the runtime. The runtime can use this protocol to restrict access to only those applications it trusts.

Message format

Every network protocol need an a priory agreement upon the format of the messages used in the protocol, otherwise the receiver will not be able to parse the message into its component message elements. In a flexible protocol the agreement could extend to using a list of possible message formats, but the list can never cover all possible message formats as that list would be infinite. Neither is it possible to detect and understand an unknown message format. To see why, think of a message as a bit sequence of length n. A bit sequence has no inherent meaning and to convey information it must be interpreted by a

protocol. However, there will always be an infinite number of possible protocols, and unless there is a priory agreement the receiver has no way of knowing which protocol the sender used.

So we need a least one message format for communication. Since many protocols have their own message formats (e.g. SSH [97], Kerberos [40], PGP [98], SPKI [27], SSL/TSL [81], SET [81], S/MIME [81]), it seems a good idea to create a standard API for programming new message formats: After all, formatting messages seems like a well defined task and it should be possible to find good abstractions. For instance, one could use formatting code in the Obol scripts; a new primitive based on Common Lisp's format or C's printf is one solution; another is to introduce procedures using a lambda primitive: The advantage with such an explicit abstraction is precisely that it makes the message formatting part of the protocol explicit. The downside is that the script becomes longer and may obfuscate the semantics of the protocol. Message drivers are another abstraction for handling message formats. They tie in with the runtime and do message formatting behind the scenes. This is an attractive solution because the Obol scripts remain small, but it hides the fact that the end programmer also has to program the message drivers; and they will have to be programmed in whatever language the runtime is implemented in. Since this thesis is about identifying proper abstractions and not about being binary compatible with other protocol implementations, we have chosen the simplest solution that still allowed us to send and receive messages over a TCP/IP network: A single, Obol-specific message format implemented as a default message driver; default means that we don't have to specify it in the Obol program, it will be chosen every time we send or receive a message.

We have chosen a simple ASCII message format where 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 will make parsing easy. Here is a message containing a 128 bits AES key. 128 bits is 16 bytes which must be encoded with 32 hex digits:

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

A message containing a 4 bytes timestamp and the same 128 bits key would look like this:

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

Note 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 has the necessary information to decode the message; decoding happens in the context of a pattern such as in **receive**; if the first element in the pattern is a nonce, the hex encoded bit sequence in the message element is interpreted as a nonce, if the pattern element is a string it's decoded as a string and so on. If this later should prove to be too weak, we could switch to a message format with more semantic information in the encoding such as SPKI [27].

encrypt

Encryption is a fairly straight forward thing to do. You use a key to encrypt one or more pieces of cleartext and return the result as a byte array. There are some challenges however. All encryption algorithms work on the bit level, typically on byte arrays which means you need a way to encode the cleartext before it can be sent to the encryption algorithm. In addition you may have to specify which encryption mode [38] you want to use and possible also an initialization vector (IV) for the mode.

The encoding used in encryption must obviously match the decoding used in decryption. Many protocols use their own message formats just as described for sending and receiving messages above. We chose our own internal message format for encryption and decryption, and we did it for the same reasons we did it for **send** and **receive**. The format is almost the same as well; we first transform the cleartext to an ASCII string delimited by parenthesis as described above and then we translate the string to a byte array using a one-to-one mapping.

(encrypt key data attributes): This expression encrypts **data** with **key.** The key can be a shared key or a public key, and determines the encryption algorithm used, e.g. an AES keys results in AES encryption. The exact format of the encryption depends on the algorithm and encryption mode we use, and we can use the optional **attributes** operand to specify the mode and other parameters like an IV. If no encryption mode is specified, the default should be used. The possible options will depend on the underlying cryptographic library Lobo is using. We want **receive** to accept a sequence of values as **data**:

```
(encrypt key value1 value2 ...)
```

The values or message elements can be Obol variables or literals. The return value is the ciphertext represented as an unsigned byte vector. This is tailored to the decryption operator which takes an unsigned byte array as one of its operands.

In the following example we first use **generate** and **believe** to generate a shared key and than we encrypt a cleartext with the key. The cleartext is given as a string literal and we see the return value is a byte array:

Example:

```
(believe K (generate shared-key ((alg AES) (size 128))))
(encrypt K “secret”) => #(89 207 5 65 106 ...)
```

Exceptions

Most modern languages define a structured way to handle runtime errors, whether they are hardware related, such as floating-point overflow, or software related such as performing an operation on values with types that makes no sense for the operation. According to Pitman [63], programming is telling the computer what to do in every circumstance, and when the program reaches a place where there are several possible next steps and is unwilling or incapable of choosing among them, it has detected an *exception*. The possible next steps are

called *exception handlers* [73] and an exception is *raised* or *signaled* when the associated event occurs.

How should exceptions be handled in Obol? Sebesta [73] lists several design issues and we will go through some of them:

- How and where are exception handlers specified? Because Obol is not a general purpose programming language, it's not obvious there should be any exception handlers specified in Obol. Instead, it could be the job of the application (written in some general programming language) to test for and handle any exceptions occurring in Obol. Alternatively the runtime could offer some predefined exception handlers that could be specified by some sort of label in the Obol program.
- How is an exception bound to an exception handler? When an exception is raised, how does it arrive at the correct exception handler? This is complicated by the fact that the runtime may well be implemented in a different language than the application that loaded the Obol script, and the application and the runtime may not be in the same address space. This indicates that the binding should use a protocol that is a) independent of the involved programming languages and b) suitable for use over network connections.
- Where does execution continue, if at all, after an exception handler completes? Basically there are two choices, either the Obol program terminates or program execution can continue. Termination is the simplest choice, both in terms of implementing Obol but also conceptually. It may also be the best; if we are waiting for 128 bits fresh nonce in a protocol message and we receive a message with a 64 bits nonce there is little we can do about it. We could of course wait for a new message, but such an error is likely to be the result of a serious programming error on the other side and we would likely never receive a new message. But if we decided to wait, we would return to the expression that raised the exception in the first place. This may work in this example, but would not be a good idea if the condition that raised the exception was still there because the exception would be raised again. So returning to the expression that raised the exception is only useful if the exception handler is able to modify the condition that led to the exception.
- Should it be possible to raise exceptions explicitly by the user? This can be useful if there are software-detectable conditions in which the user would like to signal a specific type of exception, or equivalently, conditions where the user would like to invoke a specific handler. As long as Obol is not a general programming language, it is less likely that we have the ability detect error conditions explicitly; indeed, it may be a feature of Obol that we don't have to concern ourselves with such issues. However, we feel this cannot be decided before we have more practical experience with Obol.

Based on the above discussion, we think exception handling should be left to the application, at least until we gain more experience with typical exceptional situations in Obol. We want a simple string based protocol for signaling that an error has occurred. The exception should be followed by a short description of the error condition, destined for a human reader. Because some applications may want to continue after an exception has been

raised, the runtime should be able to continue a script after it has stopped due to an exception. We want to be able to raise exceptions explicitly, at least for now. In the future, we might remove this feature if it turns out to be of little practical use.

decrypt

The **decrypt** operator has the same relationship to **encrypt** as **receive** has to **send**; it must be able to decode a message using the a priory agreed upon message format used in encryption. And like **receive**, it must be able to look for a matching pattern in the decrypted message. In this respect, **decrypt** is simply an extension of **receive**, and we even considered combining **receive** and **decrypt** in a single operator **receive-decrypt**; we didn't because the semantics of pattern matching with and without decryption is different. In **receive,** the pattern matching is used to look for a message belonging to this script, this should never be the case for decryption; following the principle of self-contained messages discussed several times already [3], it should be possible to deliver a message to the intended script without doing any decryption operations; if we have to do decryption before we can decide which script should get a message, we should redesign the protocol. This means that when a message (or to be precise, parts of a message) is ready to be decrypted, we already know that this message belongs to this script. Pattern matching after decryption is used to verify that the decryption worked; failure to find the expected pattern after decryption therefore have very different consequences than failure to match a pattern in **receive**: In **receive** we simply continue waiting for a new message, but in **decrypt**, failure to find the expected pattern is an error. It could be that the message has been tampered with, or maybe either the sender or the receiver or both are using the wrong key. Anyway we have a runtime error that should be signaled to the application and it should be up to the application to decide what to do next. The script should therefore not be allowed to continue unless it gets some kind of confirmation from the application.

(decrypt key ciphertext pattern attributes): This expression decrypts **ciphertext** with **key** and expects to find **pattern** in the decrypted message. The pattern checks that all non-anonymous variables occur in the message in the specified order and checks that the message has enough elements to fill any anonymous variables. This may leave some message elements unaccounted for and that's allowed. The order of the message elements and the pattern elements do matter. As a side effect, any new anonymous variables in the pattern are created and assigned values. They can later be promoted to typed values with **believe** or sent on unprocessed to a remote host. As **receive**, **decrypt** returns a list of values, namely the message elements found inside the ciphertext after decryption. The **attributes** operand can be used for initialization vectors and other parameters to the decryption algorithm.

We have designed **decrypt** to abort with an error message to the application if a pattern is specified but the message doesn't match after decryption. The runtime offers an API for continuing the script if the application decides that is appropriate, but we think that is unlikely to happen.

Examples:

```
;; The first example shows how encryption and decryption
;; are inverse operations and negate each other when we
;; apply them to a byte array.
(decrypt K (encrypt K #(1 2 3 4))) => #(1 2 3 4)

;; This expression tries to decrypt the first available message
;; and will abort with an error if the first element after
;; decryption isn't equal to the contents of the variable A.
(decrypt K (receive) A)
```

sign and verify

Some protocols sign part or all of their message elements: PGP is one example [15], the key negotiation protocol in [28] another. We therefore want a sign primitive with an accompanying verify primitive. As for encryption/decryption the algorithm should be determined implicitly by the key. The **sign** operator should return a digital signature on a piece of data. The v**erify** operator should verify the signature on the same piece of data using the public key corresponding to the private key that made the signature. It the signature is false the script should be aborted with an error message.

As for encryption, the data has to be encoded as a byte array before the signature algorithms can use it. We use the same encoding/decoding as for encryption/decryption. This is natural since the algorithms are often related, and indeed, in the case of RSA signatures, signing is simply encrypting with the private key and verifying is decrypting with the public key. The syntax of **sign** and **verify** follows.

`(sign private-key data1 data2 ...):` This expression signs the data with the private key.

`(verify public-key signature data1 data2 …):` This expression verifies the signature on the data.

In the following example we sign a piece of data and immediately verify it. As expected, the signature is correct and no exception is raised. The variables PUB and PRIV contain the public and private key respectively.

Example:

```
(verify PUB (sign PRIV A) A)
```

return

In simple Obol scripts there is no need for a separate primitive to terminate execution. Every expression is executed in order and if no error has occurred and we execute the last expression successfully, the program simply terminates. However, if we add loop

primitives or conditionals like **if**, we may want to terminate execution before all possible expressions have been executed.

A primitive to terminate execution also often has a second functionality, it returns values. The return primitive is placed at the end of a block of code and returns a value (or values) to the caller of the code. Again we have no need for this in Obol, because we use a functional style where every expression implicitly returns a value. However, we do need a way to return values to the application that started the Obol program. We could use a convention that said that the value returned from the last expression in the script was returned to the application. But this will be awkward when the value we want to return doesn't occur naturally in the last expression. As a consequence, we want a separate primitive for returning values.

The return value could be a simple true/false value to indicate if the protocol was successful, or maybe the session key that was the result of an authentication protocol. The runtime must offer an API to get hold of this value and in the general case when Lobo is in a different address space than the application we need a protocol for marshalling and un-marshalling the return values.

`(return result):` The basic functionality of the **return** operator is to terminate execution and return a value from the script. The runtime must make the return values available through its API for the client who loaded the script. As a special case, a single **return** expression simply exits the script without returning any values.

Example:

```
;; Return the variable Kab and abort the script. The variable
;; will be available through the runtime API.
(return Kab)
```

During prototyping we found it useful to add some variations to the basic return expression; for instance, we sometimes wanted to return a value, but continue execution. We therefore added an optional type attribute to **return**.

(return value attribute): An optional type attribute can be used to modify the behavior. The default type is "exit" and the short form **(return value)** can be used instead. The type "continue" means that a value is returned to the application, but the script continues.

Example:

```
;; Return the variable Kab, but continue the script. The
;; returned value will be available through the runtime API.
(return Kab ((type continue)))
```

Instead of adding a separate primitive for raising or signaling conditions we decided to use the **return** operator since the runtime has to do the same work in both cases. We added the type "error-exit" to signal an exception; the value operand can be used to describe the

condition. For completeness, we also added a type “error-continue” which can be used to signal a non-fatal, but still exceptional, condition to the application and continue the script.

```
;; Abort the script with an error message.
(return “An error occurred” ((type error-exit)))
```

diffie-hellman

In addition to encryption and signatures, public key cryptography can be used for key agreement [78]. The Diffie-Hellman algorithm [26] is extensively used for key agreement in protocols like SSH [95] and SSL/TLS [25, 81] and we therefore wanted an Obol operator for it. The original Diffie-Hellman algorithm can actually be used with any group where the discrete logarithm problem is hard to solve [49], so we might want the operator to cover for instance elliptic curve versions of the algorithm as well.

(diffie-hellman key): This expression will generate a random element of the group (typically an integer), do modular exponentiation using the base and modulus of the key and return the result (the “half-secret”) as per the standard Diffie-Hellman operation [26]. The random group element will be stored implicitly with the key.

(diffie-hellman key half-secret): This will compute and return the shared secret according to the Diffie-Hellman algorithm. The key must have been previously used to create a “half-secret” and **half-secret** is a “half-secret” created by *another* instance of the same key, typically the remote principal in a Diffie-Hellman key exchange protocol.

load and store

We need a way to load a priory distributed knowledge like shared keys or trusted certificates into a script. Usually, the application that owns the Obol program would load such keys from a file or database and send it to the script, but for testing purposes it would be convenient to use a **load** primitive to load something directly from a file inside the script. A **store** primitive is useful for storing keys that must survive several sessions. It can also be used to store the result of lengthy cryptographic operations like generating long RSA keys.

When loading data, **load** is basically receiving a message and we must handle the same issues common to all receive primitives in message passing. For instance, should **load** be blocking or non-blocking? We chose blocking, because we felt it would be easier to use **load** to synchronize with the application that way. There is also the question of message formatting. We simply assume messages come from the runtime and thus are readily available in the native representation of whatever language the runtime is written in. This is natural because the application has to call a function in the runtime’s API to be able to send a message to the script.

`(load source attributes):` This operator will load and return the content of the source. The main use of **load** should be to get data from the application so we make **source** optional and let it default to the application. The application must use the runtime API to send data to the script. If the source is an address, **load** should look for a key belonging to that address. The attributes are optional as always and can be used to set a timeout similar to the timeout used in **receive**.

Examples:

```
;; Loading data from a file. The data will be interpreted as
;; an AES key and stored in the variable K.
(believe K (load "K.key") shared-key ((alg AES)))

;; Load a string from the default source which is the
;; application, and store it in a variable with believe.
;; The application can use the Lobo API to send messages to
;; the script. If no messages are available, this call will
;; block the script.
(believe M (load) string)
```

`(store variable path):` The **store** operator will store the content of the variable in a file. This operator is only added for convenience, and the same functionality can be achieved by returning the value to the application and storing it there.

Extra operators

While testing the core operators by writing protocols, we found it convenient to extend Obol with loop operators like **while**, control statements like **if**, the ability to create scope and local variables, procedures and some more. For instance, it's difficult to implement a server without some kind of loop primitive, although the application could start a new script for each incoming request. We need more experience with Obol in real life settings before we can decide if we really need these extra operators, and as we stated at the beginning of the chapter, we want to keep Obol a simple language if at all possible. Therefore, we will not discuss the extra operators here, but instead present them in the next chapter.

Lobo – the Obol runtime

There are two basic alternatives for implementing a programming language; as a compiler or as an interpreter [29]. In an interpreter, Obol programs will be passed through a front end that converts it to an abstract syntax tree. The syntax tree is then passed to the interpreter which executes it and returns an answer. A compiler also takes an abstract syntax tree as input, but translates it to some other language (the target language), which in turn is executed by an interpreter. Usually, this other language is a machine language like the one in the x86 architecture, but some language implementations use a special-purpose target

language that is simpler than the original and easier to implement; Java is a typical example.

Obol, or any programming language, can be regarded as a coherent set of abstractions built on top of the machine language [4]. Interpreters are suited for interactive programming and debugging because the steps of the program execution are organized in terms of these abstractions, and therefore easier to understand for the programmer. Compiled code can execute faster because the steps of the program execution are organized in terms of the machine language, and the compiler is free to make optimizations that cut across the high-level abstractions.

We find a design suited for interactive programming and easy debugging compelling when we are designing a new language, something which involves a lot of trial and error. For instance a good tracer will be useful when we are testing the syntax and semantics of new operators, and interpreters make them easy to write [65]. We therefore decided to implement Obol as an interpreter. Since our main goal is to identify proper abstractions, we are not concerned about speed at this point in time, and we could always implement a compiler later on.

The job of an interpreter is to take an expression and an environment as input, evaluate the expression in that environment and return the result [5, 29]. If we use the interpreter as an interactive prompt, it should behave like a read-eval-print loop (REPL); that is, read the expression, evaluate it, print the result to an output stream and loop to do the same again. Because a top-level expression may contain nested expressions, a recursive structure is natural for the evaluation part. Since we have designed Obol to always begin with an operator, the core of the evaluation will be a switch on the first symbol in the expression. If we add a parser to this design, we have a complete design for an interactive Obol-prompt. Such a prompt will be useful when experimenting with the operators of Obol.

When we run a protocol, we normally don't want to see the output, we simply want to load a script and run it. Therefore, Lobo should have an interface which parses a script, loads it into the runtime and runs it; we call this interface **load-script**, and it must return a handler to the script. The client can use the handler to send messages to the script with **send-msg-to-script** and get values returned from the script with **get-result**. We also add a call **continue-script** for waking up a script that has stopped, either because it has blocked waiting for a message or because it has aborted due to an error. This call can be used to continue a script after the error has been fixed. Together, these calls constitute an interface to Lobo that can be used when Lobo is running in the same address space as the client. From the client's point of view, Lobo is simply a linked library.

If Lobo is used as a linked library, there is an implicit secure channel between the client application and the runtime. If the client is not in the same address space as Lobo, we need a different kind of secure channel and some sort of remote procedure call (RPC) mechanism. For instance, assume the client is a smart card. A smart card has limited resources and may not be able to run Lobo itself. Instead it might want to contact a trusted Lobo instance and use the instance to execute Obol programs on its behalf. This will

involve the following steps: First the client and Lobo must run a bootstrap protocol to authenticate each other and set up a secure channel. Next the client loads a script over the newly created channel and Lobo executes the script. Then Lobo returns the result of the script to the client. The return value depends on the nature of the script and could for example be a simple Boolean, a key or a secure channel to a remote host. Finally, the client may invoke operations on the returned value; for instance it may use a secure channel to send data to a remote host.

To design and implement a fully functional robust and general RPC mechanism was beyond the scope of this thesis, but we designed a simple RPC like protocol to see how a RPC mechanism would affect the design of Obol. We won't go into details, but the overall design was as follows: Lobo has a server that listens for incoming RPC connection requests. When a client sends a request, they run a bootstrap protocol. After that, the client sends the script and the RPC-server in the runtime (or server side stub if you will) uses the standard Lobo API to load the script. On the client side we need a proxy that could understand the ASCII based protocol spoken by the Lobo side RPC server. The proxy offers an API based on what it receives from Lobo. The server side stub communicates with the Obol scripts using the standard Lobo API, namely **load-script**, **get-result** and **send-msg-to-script**.

As it turned out, Obol remained mostly unchanged after adding the RPC mechanism and we took that as a good sign. But the exercise forced us to think through the semantics of **return**; for example, when we return a shared key after a successful authentication protocol, what should the application see? We could return the key as raw bits and leave it to the application to find a use for it; for instance, it could send it back to Lobo as input to a script that realizes a secure channel, or the application could open a channel (typically a TCP connection) and use the key to realize a secure channel itself. But then the application needs access to a cryptographic library which can use the key and we don't really want that because it means that security related code is leaking back into the application, precisely what we set out to avoid. Alternatively, the runtime could return a proxy with operations like send and receive, in essence a handle to a secure channel. We decided to try both solutions.

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 it encounters a blocking operation, like a **receive** with no matching message in the in queue. Then it yields to the next active script. In other words, we are not using preemption. Other events in the system are also synchronized by going through the main event queue. We could have chosen threads for multiplexing the scripts. Then it would have been natural to start a new thread for each script. However, it would have forced us to explicitly synchronize many of the data structures in the runtime. We therefore judged an event based system to yield the simplest design, and we indeed had very little trouble with synchronization during the development.

A downside to not using preemption is that a busy script will starve the other scripts. However, we expect Obol scripts to be short and spend most of their time waiting for

messages or doing low-level encryption. Waiting for messages will not be a problem because **receive** will yield. Doing low level cryptographic operations can potentially take a long time, and we could have added a mechanism for yielding after an allotted time slice had been spent. Threads would handle this automatically. However, the overall throughput of the system would be reduced because of the costs of doing context switches [87]. None of this mattered much to us though, because we were designing a prototype and performance was not an issue.

Conclusion

We believe the design presented above fulfills the requirements from the previous chapter. The design is for a research prototype, not an industrial strength product, and we have therefore allowed ourselves to do some simplifications. For instance, we assume Obol programs will behave “well” and not claim all available resources forever. As a consequence, we have no mechanism for preemption in the runtime.

5 Implementation

We chose to implement Lobo in Common Lisp using Allegro 6.2 because it offered a high-level language with a powerful environment which simplified the implementation. And we can verify MacLennan's statement that it's smart to implement an interpreter for a language with a Lisp like syntax in Lisp [46]; Obol's prefix syntax with parenthesis and a functional oriented semantics made Common Lisp a good choice. We used the CLC cryptographic library [74] to handle low level cryptographic operations, because that was the only cryptographic library available for Common Lisp.

The implementation consists of about 3000 lines of code. This may not seem like much, but the high-level, functional oriented nature of Common Lisp combined with the very compact programming style that is the convention for writing Lisp programs, mean that the code contain relatively much functionality. For example, the body of the complete Obol parser can be expressed in just three lines. We believe it would take considerably more code to write an interpreter with the same functionality in for instance Java.

Syntax and semantics

We have implemented all the operators according to the syntax and semantics described in the previous chapter, which in turn fulfills the requirements from chapter 3. In the following, we will comment on some of the implementation details for the operators.

believe

If **believe** fails, the script creates an exception and aborts. The application can detect this by using **get-result**. Currently, **believe** do the following checks which all may result in an error:

- If the expression has a size attribute for shared-key types, the expression fails if the value contains too few bits, for example if we try to interpret a 128 bits value as a 256 bits key.
- Timestamps (represented as seconds from an arbitrary starting point) are checked against the local time and deemed valid if they are less than the current time + an optional lifetime (clock skew).
- Nonces are checked for freshness. This is handled by storing every nonce the script sees and comparing every new nonce against the list of previously stored nonces.

We illustrate the three checks with three examples which all transform a value in an anonymous variable into a new typed value stored in a variable:

```
;; Interpret the contents of *K as a 256 bits AES key.
;; The value in *K must be 256 bits or an exception will
;; be raised.
```

```
(believe K *K shared-key ((alg AES) (size 256)))

;; Interpret the contents of the anonymous variable *T as a
;; timestamp and verify it with a lifetime (clock skew) of
;; 360 seconds. If the timestamp is too old, an exception
;; is raised.
(believe T *T timestamp ((lifetime 360)))

;; Interpret the content of the anonymous variable *1 as
;; a 128 bits nonce. An exception will be raised if this
;; is not possible.
(believe N *1 nonce ((size 128)))
```

generate

Valid types are `shared-key`, `public-key`, `signature`, `timestamp` and `nonce`, but the specific algorithm names and parameters like key size depend on the underlying cryptographic library, in our case the CLC library [74]. We give some examples below. Note how the generation of public keys (including signatures) produces two keys, the public and the private one. We therefore have to create two variables with **believe**.

Examples:

```
(generate nonce ((size 128)))
(believe (PUB PRIV)(generate public-key ((alg RSA) (size 256))))
(believe (PUB PRIV)(generate signature ((alg DSA))))
(generate timestamp ((lifetime 3600)))
```

send

Consider the following **send** expression:

```
(send "tasko.stud.cs.uit.no:9000" A B Na)
```

The port number used in the receiver must be the listening port of the remote Lobo instance, and currently the only way to send messages is over TCP. Messages are formatted using the internal message format. We have not implemented a mechanism for choosing another message format (because we only have one), but the current message format is implemented and used by calling two functions, one for encoding and one for decoding. By adding an attribute to **send** and **receive**, it should be fairly easy to switch between the available message formats, for example:

```
(send "tasko.stud.cs.uit.no:9000" A B Na ((format PGP)))
(send "tasko.stud.cs.uit.no:9000" A B Na ((format SSH)))
```

receive

Multiple channels (given as a “channel” attribute) can be used to specify that we are waiting for a message from one of several senders. The following example waits for a message containing 2 elements coming over either channel A or B:

```
(receive *1 *2((channel A B)))
```

In theory the channel attribute could also be used to specify multiple delivery channels, for instance TCP and SMTP, but TCP is the only incoming channel we have at the moment. The channel operand could be extended to a little query language in itself, because we might want to say something like “accept every message from A or B which comes over SMTP (but not over TCP) or any message from C over TCP”. However, we didn’t try to implement that because we felt we needed more practical experience with Obol before we could design the language and more importantly, decide if such a language was really necessary.

encrypt

The default mode for shared key encryption is cipher-block-chaining (CBC) [72], and since the underlying cryptographic library [74] doesn’t offer support for any other secure [28] mode, it’s not possible to choose anything else. For public-key encryption, the only supported algorithm is RSA, again because of the library.

return

As mentioned before, it’s not obvious what the semantics of **return** should be when seen from the application. If we return a key, what does the application get? The raw bits? A handle to a secure channel? We’ll discuss this when we describe the runtime below.

diffie-hellman

We only implemented standard Diffie-Hellman using the group Z/pZ because the cryptographic library didn’t support any other groups like elliptic curves.

load and store

When implementing security protocols one often needs to locate a key associated with one of the principals in the protocol. Therefore, if the source operand in **load** is of type address, **load** looks for a key associated with the address. Currently, this is done with a static lookup table. In a more general solution, this should be a database lookup in a database specified by the application. In the following example, the load expression will try to find a key belonging to the address “gaupe.cs.uit.no”.

Example:

```
(believe A “gaupe.cs.uit.no” address)
(load A)
```

Extra operators

As explained in the design chapter, we extended Obol with some extra operators. We consider them to be experimental extensions and not a part of Obol yet, but we present them for completeness.

`(lambda parameters expression):` The **lambda** primitive is a normal lambda operator per lambda calculus and can be used to declare functions in combination with **believe**. The addition of a lambda primitive also makes Obol Turing complete [90]. However, this does not mean that Obol becomes a general purpose programming language; as mentioned in the theory chapter, using lambda for general calculations is not very practical. The following example shows how we can declare a function.

Example:

```
;; Note that we had to add a subtraction primitive as well to
;; make this example work.
(believe foo (lambda (x) (- 3 x)) function)
(foo 4) => 1
```

`(format control-string &optional args):` The **format** operator is used to create formatted output with control syntax equal to Common Lisp’s format primitive [82]. There is also a **print** operator for doing unformatted output. The following example prints a string to the Obol script’s standard out and inserts the string representation of a variable into it.

Example:

```
(format “Value of variable is=~A.” var) =>
“Value of variable is=#X7228e462fe75575439a06ee59d1ad3c5.”
```

`(let ((var value)*)  body)`

The let primitive corresponds to the let form in Common Lisp [82] and can be used to create a scope with local variables. The following example creates two local variables “x” and “y” and subtracts y from x. The **let** expression as a whole will return the value 2.

Example:

```
(let ((x 6)
      (y 4))
  (- x y))
```

`(while test body):` This expression is a standard while primitive.

`(if test then-form else-form):` This expression is a standard if primitive.

`(progn form1 form2 …):` This expressionis a primitive for creating a block. It has the same semantics as the Common Lisp progn primitive [82].

`(list var1 var2 …):` This expression is a primitive for creating a list and is useful in any language that uses lists a lot.

`(nth n list):` This operator extracts the nth element from a list and is useful in combination with primitives that return lists, for instance **decrypt**.

`(bind var value):` This operator creates a new binding [65] for a variable inside a local scope like a **let** expression. This is different from **believe** which has dynamic scope.

(- value1 value2): This expression subtracts **value2** from **value1** and returns the result. It works for integers and byte arrays. For bytes arrays we first interpret them as integers, do the subtraction and convert back to a byte array again. The reason for accepting byte arrays is that the Needham-Schroeder protocol uses this operation to modify a byte array.

Parsing

Choosing a parenthesis based structure for Obol made parsing simple; we could use the Lisp parser in the Common Lisp primitive read. To illustrate this, the following shows the complete parser in Common Lisp:

```
(loop for exp = (read stream nil nil)
        until (eq exp nil)
        collect exp)
```

Lobo – the Obol Interpreter

The interpreter has been implemented as described in the design chapter and fulfills the requirements from chapter 3. We present some of the details from the implementation.

The Obol read-eval-print loop (REPL) was implemented as described in the previous chapter. The function **evaluate** is at the heart of the evaluation. The following Lisp code shows part of the case expression which dispatches on the operator:

```
(defun evaluate (exp script-env local-env str)
…
 ((case (first exp)
…
      (believe (eval-believe exp script-env local-env str))
      (generate (eval-generate exp script-env local-env str))
      (encrypt (eval-encrypt exp script-env local-env str))
      (decrypt (eval-decrypt exp script-env local-env str))
      (receive (eval-receive exp script-env local-env str))
…
```

The simple nature of the design makes it easy to extend Obol with new primitives; all you have to do is insert a line in the case expression and write a corresponding eval-function. There is no need to rewrite the parser because every Obol statement is a list surrounded by parenthesis where the first element of the list is the operator and the rest are operands.

Note how **evaluate** takes a script environment and a local environment as input in addition to the expression itself and an output stream associated with the script. Normal Obol variables are stored in the script environment and they have “script global” scope. Local variables with local scope created by the optional **let** expression live in the local environment. The environment associates values with variables, and we have chosen the conventional data structure for representing such associations in Lisp [65], namely association lists [82].

As mentioned under the description of **format**, all output from a script is sent to the stream belonging to the script and can later be viewed by **get-script-stream** if the script was loaded in a non-interactive mode. We did this to make it easy to redirect the output to different targets. For instance, when using a prompt, we obviously want the output sent to the screen, but the screen could mean several things. In our case it meant the Common Lisp prompt or a telnet terminal connected to Lobo via a simple telnet server we added to the runtime. The telnet terminal functioned as a simple GUI, and by starting two terminals, each connected to a separate Obol prompt, we could simulate the two sides of a protocol by hand.

If the interpreter encounters an error in the interactive prompt we would normally be aborted from the interpreter with a fatal error, but after working for a while and having accumulated a lot of state, it would be nice to be able to fix the error and continue without loosing the state. We therefore added a break-loop [82]. A break-loop would normally be a read-eval-print loop similar to the top level Obol prompt, where the user types expressions or top-level commands, causing Lobo to evaluate the expression or carry out the command. However, we replaced it with the Common Lisp top level because it was more useful during development and debugging. We kept some Obol specific functionality, including the Obol trace leading to the error. The break loop begins when an error occurs in the interactive prompt, and it displays the trace of the operators with their arguments leading to the error and has the opportunity to replace an incorrectly formed expression with the correct syntax and restart the evaluation. This was pretty useful because the syntax changed

a lot during development and it was easy to use old syntax by mistake. We illustrate the use of the break loop in the next chapter.

When a client application loads a script with **load-script**, an integer is returned:

```
(load-script “Needham-Schroeder.obol”) => 45
```

This integer becomes the handle to the script and can be used by the rest of the Lobo API to communicate with the script. As an example, **get-result** uses the handle to get the value returned from a **return** expression. The handles should normally be large, randomly generated integers to avoid malicious applications guessing the handles of scripts belonging to other applications, but since we assumed well-behaved applications elsewhere, we made the same assumption here and used small integers to ease manual testing.

The example with **get-result** leads us to another issue. What precisely does a **return** expression return? What does the application see? As default we return raw bit values formatted in the internal package format used by **send/receive**. When the application is using Lobo as a linked library, it can use this message as input to a new script (typically a script realizing a secure channel) or use library functions from the Lobo implementation to decode the message if it wants to interpret it. As an alternative, we also implemented a RPC mechanism which presents a more abstract interface to the returned value. However, the RPC implementation is very limited and must be taken as a proof of concept. We need more hands on experience with Obol in real life settings before we can come up with a good design and implementation for this mechanism. As for now, the mechanism can only be used to return a shared key as a secure channel. The application must implement a proxy that understands the RPC protocol. The application initializes the proxy with an authentication script, and behind the scenes the proxy sends this script to Lobo, which runs the protocol. If the protocol is successful, Lobo makes two methods available for the application via the proxy: send and receive, and they can be used to communicate with the remote host.

The **send** and **receive** primitives need support from Lobo to communicate with remote Lobo instances. We designed a packet server that listens for incoming packets on a predetermined port. The packet server store incoming packets in an in queue and unblock any blocking scripts. Each script will look at the available packets and accept (and consume) the first matching packet or block if none of them match. The **send** primitive correspondingly creates a connection to a remote packet server port and sends the packet.

Central to the event based architecture of Lobo is the ability to store state for each Obol script. The state is stored in instances of the **ObolScript** class and includes the environment, the output stream, the current expression being evaluated, timers and more. When a script yields to another script, for instance as the result of blocking on a **receive** expression, the current state of the script is stored and put into a queue of active scripts. The scheduler then picks the next active script and runs it until it blocks and so on. There is no preemption so a script that enters an infinite loop will block Lobo. We allowed this because we assume the scripts “behave well”, and besides we consider loop primitives to be an experimental feature that might not make it to a later version of the language.

The event scheduler handles the timeout mechanism in **receive**. As explained when we described the operator, a call to **receive** has an optional timeout attribute that sets a timer which time out after the given number of seconds. Before a script is started (or restarted after a block), the scheduler checks the timer and if it has timed out, the script is marked as "done" and not considered for scheduling again.

Conclusion

We have implemented a runtime for Obol which runs Obol programs in an interpreter. The runtime fulfills the requirements set forth in chapter 3 and is implemented according to the design from the previous chapter.

6 Testing

We will demonstrate the interactive Obol prompt, show Obol implementations of several protocols and present an application that uses Lobo.

The interactive prompt

The interactive Obol prompt is useful for testing the functionality of the various primitives and can also be used to run a protocol step by step. The prompt can be started as a separate process with **obol** from the Common Lisp top-level, or we can connect to a running Lobo using telnet. We illustrate the first method:

```
lobo(432): (lobo)
Starting Lobo...
Opened port 9000 for packet server.
lobo(435): (obol)
obol[1]>
```

We now have an Obol prompt connected to a script with id 1. To generate a 128 bits AES key and store it in the variable K we do:

```
obol[1]>(believe K (generate shared-key ((alg AES)(size 128))))

<ObolVariable:
    size 128
    value <clc:SymmetricKey
value:#Xdcfc1454eef77b68c2a29e39c21899b3>
    alg AES
    type shared-key>
obol[1]>
```

To load the contents of a file we call **load**. Note how the content is returned as a byte array because **load** attaches no meaning to the data.

```
obol[1]>(load "K.key")

      #(7 249 178 209 58 128 250 203 80 157 144 244 44 22 50 106)
obol[1]>
```

To attach meaning to the value loaded from the file we must use **believe**.

```
obol[1]>(believe K (load “K.key”) shared-key ((alg AES)(size 128)))

<ObolVariable:
    size 128
    value <clc:SymmetricKey
value:#X07f9b2d13a80facb509d90f42c16326a>
    alg AES
    type shared-key>
obol[1]>
```

To send an encrypted message (in this case an array with four bytes) to a remote Lobo instance we use **encrypt** and **send**:

```
obol[1]>(send "nb75.stud.cs.uit.no:9000" (encrypt K #(1 2 3 4))

"((72c448d17499de537f30fb3adb9c63e6))"
obol[1]>
```

Note how the **send** expression returns the message encoded in the internal message format. To receive and decrypt that message on the remote Lobo instance we do:

```
obol[1]>(decrypt K (receive))

#(1 2 3 4)
obol[1]>
```

If the interpreter encounters an error we have the opportunity to fix it in the break-loop and then continue. The following example illustrates the break-loop; suppose we want to generate a nonce, but forget to add the **nonce** type specifier to **generate**:

```
obol[1]>(generate ((size 128)))

------------------------------------------------------------------
--
Error: Unkown type in generate:((size 128))

Restart actions (select using :continue):
 0: Replace expression '(generate ((size 128)))'.
 1: Abort entirely from this process.

[changing package from "common-lisp-user" to "lobo"]
[Current process: evl]
[1] lobo(1): :continue 0
New Obol expression (old='(generate ((size 128)))'): (generate
nonce ((size 128)))

------------------------------------------------------------------
--

<ObolVariable:
    size 128
    value #(102 76 241 3 97 29 182 14 80 204 1 182 55 3 173 186)
    type nonce>
obol[1]>
```

Inside the break-loop we chose to replace the erroneous expression by typing **:continue 0**, and when we entered a correct expression, we was returned to the prompt as if the error hadn’t happened.

Protocols

We have chosen to present four protocols. The first one, wide-mouthed-frog, was chosen because it was short, the two next, Needham-Schroeder and PGP, were chosen because they are famous and widely used. The last, EKE, has desirable properties like forward secrecy and uses public key encryption a lot. We have used an abstract description as basis when implementing them. Therefore, they don't follow the explicitness principle from [3] so the messages are not self-contained and it would likely end in confusion if we tried to run multiple instances of these protocols together on the same runtime.

Wide-mouthed-frog

The Wide-mouthed-frog is a short authentication protocol that establishes a shared key between A and B with the help of a trusted server S [21].

Message 1 A->S: A, {Ta, B, Kab}Kas
Message 2 S->B: {Ts, A, Kab}Kbs

The protocol is flawed [22], but is still useful for introducing Obol. Note that we are not using real network addresses in this first example. This makes it easy to run all the principals in a protocol on the same runtime.

The Obol script for A:

```
1:   (believe Kas (load "Kas.key") shared-key ((alg AES) (size 128)))
2:   (believe A "A" address)
3:   (believe B "B" address)
4:   (believe Kab (generate shared-key ((alg AES) (size 128))))
5:   (believe Ta (generate timestamp))
6:   (send "m1" A (encrypt Kas Ta B Kab))
```

Line 1 starts by loading an a priori distributed shared key between the server and A into the variable Kas. The **load** statement assumes the key is stored in a known key format. The use of **believe** signals that A trusts it. If the value from the file doesn't fit the requirements, for example if the key is too short, the statement fails and the script aborts. Note how the functionality of this line is hidden in the high level description. Line 2 and 3 store the addresses of A and B in two variables; as mentioned, we don't use network addresses. This is convenient when testing a protocol and we do the same in line 6 when we use the message id ("m1") as address to **send** and not a network address. Line 4 generates a new 128 bits AES key and stores it in the variable Kab. Line 5 generates a timestamp Ta set to the current time and line 6 encrypts a message with three elements using the key Kas and sends A's address and the encrypted message to the server S.

The Obol script for S:

```
1:   (receive *1 *2 ((channel "m1")))
2:   (believe A *1 address)
3:   (believe Kas (load A) shared-key ((alg AES) (size 128)))
```

```
4:   (decrypt Kas *2 *3 *4 *Kab)
5:   (believe Ta *3 timestamp ((lifetime 3600)))
6:   (believe B *4 address)
7:   (believe Kbs (load B) shared-key ((alg AES) (size 128)))
8:   (believe Ts (generate timestamp))
9:   (send "m2" (encrypt Kbs Ts A *Kab))
```

At the server, we receive the message in the first line and use **believe** in the second line to try to convert the first message element to an address. If **believe** fails, the expression throws an exception and the script aborts. Assuming all went well, we use the address to find a key representing A in the third line. We then use the key to decrypt the rest of the message in line 4. After decryption we expect to find three message elements (*3, *4 and *Kab) and in line 5 we try to convert the first of these to a timestamp. We assume the timestamp is valid if it's less than one hour old. In line 6 we convert the contents of the anonymous variable *4 to an address and use it to load a key in the next line. Finally we generate a new timestamp and send the message to B on the internal channel "m2".

The Obol script for B:

```
1:   (believe Kbs (load "Kbs.key") shared-key ((alg AES) (size 128)))
2:   (decrypt Kbs (receive ((channel "m2"))) *Ts *A *Kab)
3:   (believe  Ts *Ts timestamp)
4:   (believe A *A address)
5:   (believe Kab *Kab shared-key ((alg AES) (size 128)))
```

At B we receive the message from S and verifies the content using **believe** expressions. A and B now have a shared key they can use to communicate.

It's not obvious that we have implemented the protocol correctly, and the same point holds for the other protocol implementations we will present as well. However, we believe it's easier to convince oneself that the implementation is correct in Obol than doing the same with an implementation in say Java.

Our implementation of the Wide-mouthed-frog protocol certainly has some simplifications. We didn't use real network addresses and ran all the scripts on the same Lobo runtime. Using channels ("m1", "m2" etc) in **send** and **receive** enabled us to avoid any problems related to routing of messages like which script gets which message. We illustrate real addresses next with an implementation of the Needham-Schroeder protocol.

Needham-Schroeder

We repeat the Needham-Schroeder shared-key authentication protocol presented in the theory chapter:

> Message 1 A->S: A, B, Na
> Message 2 S->A: {Na, B, Kab, {Kab, A}Kbs }Kas
> Message 3 A->B: {Kab, A}Kbs

Message 4 B->A: {Nb}Kab
Message 5 A->B: {Nb-1}Kab

We ran the protocol as three scripts, each script running on a separate computer in a separate Lobo runtime.

The Obol script at A:

```
 1:   (believe Kas (load "Kas.key") shared-key ((alg AES)))
 2:   (believe A "gaupe.cs.uit.no:9000" address)
 3:   (believe B "nb75.stud.cs.uit.no:9000" address)
 4:   (believe S "tasko.stud.cs.uit.no:9000" address)
 5:   (believe Na (generate nonce ((size 128))))
 6:   (send S A B Na)
 7:   (decrypt Kas (receive) Na B *Kab *toB)
 8:   (believe Kab *Kab shared-key ((alg AES)))
 9:   (send B *toB)
10:   (decrypt Kab (receive) *Nb)
11:   (believe Nb *Nb nonce)
12:   (send B (encrypt Kab (- Nb 1)))
13:   (return Kab)
```

In the last line we return the shared key with the **return** primitive. The result is available to an application or the Lisp top-level by calling **get-result** with the id of the script as parameter:

```
cl-user(1): (load-script “needham-schroeder-a.obol”)
1
cl-user(2): (get-result 1)
("OK"
<ObolVariable:
   value <clc:SymmetricKeyvalue:#X8ad33f4114621547956f8b361d1a0bb5>
    alg AES
    type shared-key>)
```

Note the subtraction operator in line 12. That operator could have been any function and examples like this would have forced us to make Obol a general purpose programming language if our goal had been binary compability with all existing and future implementation of every security protocol. But as we have said before, that was not the case and we instead implemented an interpreter which was easy to extend; all we had to do was add a line in the case expression of the function **evaluate** and write the subtraction function. And since we used Common Lisp, we could change the behavior of a running Lobo instance simply by loading the new function and reloading **evaluate**. In general, we found it very useful to have an interpreter that was easy to modify when exploring the design of a new language. By the same token we found it convenient to have a simple parser that didn’t have to be modified every time the language changed.

The Obol script at S:

```
 1:   (believe Kas (load "Kas.key") shared-key ((alg AES)))
```

```
 2:   (believe Kbs (load "Kbs.key") shared-key ((alg AES)))
 3:   (receive *1 *2 *3)
 4:   (believe A *1 address)
 5:   (believe B *2 address)
 6:   (believe Na *3 nonce)
 7:   (believe Kab (generate shared-key ((alg AES) (size 128))))
 8:   (send A (encrypt Kas Na B Kab (encrypt Kbs Kab A)))
```

The Obol script at B:

```
 1:   (believe Kbs (load "Kbs.key") shared-key ((alg AES)))
 2:   (decrypt Kbs (receive) *Kab *A)
 3:   (believe A *A address)
 4:   (believe Kab *Kab shared-key ((alg AES)))
 5:   (believe Nb (generate nonce ((size 128))) )
 6:   (send A (encrypt Kab Nb))
 7:   (decrypt Kab (receive) (- Nb 1))
 8:   (return Kab)
```

PGP

The PGP protocol illustrates the use of public key cryptography and the primitives **sign** and **verify**. The essence of the PGP protocol is listed below. A wants to send a message to B. Both A and B send their public keys to a keyserver S and we assume A obtains B's public key in a secure way, for instance from the keyserver or directly from B. The first message is A sending his public key to the keyserver. B does the same (not shown). Then in message 2, A encrypts his cleartext message to B with a temporary symmetric key, encrypts this key with B's public key and sends both encrypted elements to B:

Message 1 A->S: {Ka, A}Ka-1
Message 2 A->B: B, {Kab}Kb, {{msg}Ka-1}Kab

The Obol script at A becomes:

```
 1:   (believe K512a (load "K512a.key") public-key ((alg RSA)))
 2:   (believe K512a-1 (load "K512a-1.key") private-key ((alg RSA)))
 3:   (believe A "gaupe.cs.uit.no:9000" address)
 4:   (believe S "tasko.stud.cs.uit.no:9000" address)
 5:   (send S A K512a (sign K512a-1 K512a A))
 6:   (believe B "nb75.stud.cs.uit.no:9000" address)
 7:   (believe K512b (load "K512b.key") public-key ((alg RSA)))
 8:   (believe Kab (generate shared-key ((alg IDEA) (size 128))))
 9:   (believe msg "Secret Message" string)
10:   (send B B (encrypt K512b Kab) (encrypt Kab msg (sign K512a-1 msg)))
```

The Obol script at the keyserver S:

```
 1:   (receive *A *Ka *signature)
 2:   (believe A *A address)
 3:   (believe Ka *Ka public-key ((alg RSA)))
 4:   (verify Ka *signature Ka A)
```

The keyserver verifies A's signature in line 4. In a real life application we would probably also return the address of A and the key to the application that loaded the script for storing in a database

The Obol script at B:

```
1:   (receive *B *KabEnc *msg)
2:   (believe K512b-1 (load "K512b-1.key") public-key ((alg RSA)))
3:   (decrypt K512b-1 *KabEnc *Kab)
4:   (believe Kab *Kab shared-key ((alg IDEA) (size 128)))
5:   (decrypt Kab *msg *cleartext *signature)
6:   (believe cleartext *cleartext string)
7:   (believe K512a (load "K512a.key") public-key ((alg RSA)))
8:   (verify K512a *signature cleartext)
9:   (return cleartext)
```

We can take a look at the result by using **get-result**:

```
1:   lobo(6): (get-result 1)
2:   ("OK"
3:   <ObolVariable:
4:         type string
5:         value "Secret Message">)
6:   lobo(7):
```

EKE

EKE (Encrypted Key Exchange) was proposed by Bellovin and Merrit [18] 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 Kt 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->B: A, {Kt}P
Message 2 B->A: {{Kab}Kt}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 Kt; the attacker cannot get to the session key Kab because it is still encrypted with Kt and the corresponding private key needed for decryption has hopefully been destroyed by A.

The Obol script at A:

```
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))) shared-key ((alg
     AES)))
```

The Obol script at B:

```
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)))
5:   (believe Kab (generate shared-key ((alg AES) (size 128))))
6:   (send A (encrypt P (encrypt Kt Kab)))
```

The Secure Chat application

To understand how Obol/Lobo could interface with an application we developed a simple secure chat application which authenticates the users to each other and encrypts the conversation. A short example session is shown below:

```
1:   secure-chat[a]>Hi Bob!
2:   secure-chat[a]>
3:   Hi Alice!
4:   secure-chat[a]>What's up Bob?
5:   secure-chat[a]>
6:   I'm hacking together some cryptography
```

The application starts by loading an authentication script into Lobo. This script is used to establish a shared key with the remote chat application the local user wants to talk to. The authentication script should be a normal authentication protocol and we used Needham-Schroeder. When the script has finished both sides have the same key that represents a secure channel. To actually be able to use the channel we need to run a secure channel protocol [28], so the application loads two new scripts, a send script and a receive script. The send script gets input from the application/local user and sends it to the remote user. The receive script accepts incoming messages from the remote user and delivers them to the application/local user.

The application gives the address of the remote host and the session key to both the send and receive scripts. The send script goes into a loop accepting messages from the application using **load**, encrypting them and sending them of to the remote user. To implement the loop we used the experimental **while** primitive. The symbol "t" used in the while-test represents the Boolean value true:

```
1:   ...initialize...
```

```
 2:   (while t
 3:       (send Remote (encrypt K (load))))
```

The application uses the runtime API call **send-msg-to-script** to send chat messages from the user to the **load** expression in line 3. The receive script does the same in reverse; it enters a loop and accepts messages from the remote server, decrypts them and delivers them to the application:

```
 1:   ...initialize...
 2:   (while t
 3:       ...
 4:       (receive *msg ((channel Remote)))
 5:       (return (decrypt K *msg) ((type continue)))
 6:       ...))
```

The application uses **get-result** to get the decrypted message returned in line 5.

To demonstrate some of the power of the Obol approach suppose the application wants to change encryption algorithm or maybe do a rekey because the current key has been used to long [28]. All it has to do is to signal the other side that they should do a change, kill the current send and receive scripts, and load new ones with the new algorithm. We implemented this, and changing algorithm from IDEA to AES looked like:

```
 7:   secure-chat[a]>::change-algorithm-to AES
 8:   secure-chat[a]>
 9:   Good idea to change from IDEA to AES Alice!
10:   secure-chat[a]>Yes, I like AES better.
11:   secure-chat[a]>
```

The entire application was realized in 200 lines of Common Lisp code, a 15 lines Obol script to implement the authentication protocol and the two tiny send and receive scripts (about 6 lines each). And again, if we wanted to change the security part of the application, all we had to do was change the Obol code; the application itself would remain unchanged.

The chat application uses Lobo as a linked library. Alternatively it could have connected over TCP using the simple ASCII based RPC protocol described in previous chapters; the code on the accompanying CD ROM includes an example using this solution. Because we used Lobo as a linked library, we obviously had to implement the application in Common Lisp as well; with the RPC approach the chat application can be implemented in any language.

Conclusion

The testing demonstrates the functionality described in the Requirements and Design chapter. We therefore feel reasonably sure that the implementation fulfills the requirements we had to Obol and its runtime.

7 Discussion

We will begin by looking at the advantages of the Obol-Lobo approach, go on to discuss possible applications, weaknesses and future work and finally look at some related work.

Advantages of the Obol-Lobo approach

Reduces the gap between design and implementation

We believe Obol reduces the gap between design and implementation of protocols. Assume we have an implementation for an attractive (secure) protocol. It's impossible to guarantee that the implementation itself has not introduced new weaknesses. Ryan and Schneider [70] give an example where the implementation of a protocol with a formal security proof introduced algebraic identities which could be exploited in an attack. Another recent example is the design flaw rendering 802.11 WEP insecure because a cryptographic algorithm (RC4) was used incorrectly [84]. By implementing the protocol in a high level language closer to the design specification, we believe we can reduce the number of errors resulting from the semantic distance between the two levels of abstraction. This is not a radical idea, indeed it's the same argument used for high level languages in software engineering in general and is also one of the main advantages of a domain-specific language [76].

High level abstractions

Crypto functionality should be provided at the highest possible level in order to avoid misuse [32]. We feel that a language for writing security protocols is a step in the right direction because it allows the user to work with high level protocol primitives instead of a low level cryptographic library and a standard programming language. This also makes it easier to involve domain experts (e.g. cryptographers) in code walkthroughs because they can read the implementation in a notation closer to what they normally use instead of sifting through a lot of code written in a general purpose language. Another benefit of using a high level language is that the number of code lines in the implementation is reduced. This is good because the number of bugs increases with the number of code lines [47].

Centralizes security functionality

Obol combined with the Lobo runtime makes it possible to centralize security functionality. Instead of doing a bit of security in every application, it can all be implemented in one place. Possible examples include middleware platforms and operating systems. The central implementation obviously has to be of the highest quality, but this is the case anyway and will be easier to achieve with a centralized solution because it's simpler to find an expert once than to find many experts many times. A central implementation can also make

upgrades easier. This can be illustrated by an example: Upgrading the OpenSLL cryptographic library [92] in the Gentoo Linux distribution meant that all applications using OpenSSL had to be recompiled [52]. If the applications had implemented their security as Obol programs using the service of a central Lobo runtime, recompiling would not have been necessary.

Reflection

We believe Obol is a way to achieve reflection in the security architecture of a system. A system is reflective when it's able to manipulate and reason about itself [42, 75]. Reflection can be used to upgrade, change and add functionality to a system. This is useful in many settings: Middleware platforms can benefit from using reflection [42], transport protocols can be updated using mobile code [59], the PAM framework [71] for changing and adding authentication modules is based on this idea etc. As pointed out in [7], Obol can be used to implement reflective security in a middleware setting, and we illustrated reflection when we changed encryption algorithm in the Secure Chat application in the previous chapter. Obol fits the reflection paradigm because a system using Obol can change many security properties like key size or which authentication protocol to use, by simply loading a new Obol program. We believe a reflective security architecture is valuable in any system because security protocols and their supporting cryptographic primitives frequently change due to insecure designs and implementation errors.

Applications

Obol and middleware

General middleware environments offer an execution environment for components. This environment is usually called a capsule, or a container. Components implement the business logic of applications, while the container provides logging, communication facilities, security etc. This programming 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. The problem is the complexity inherent in security protocols. 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 SSLv2 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 implement the new protocol as part of the client and then redeploy. 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.

The solution proposed in [7] is to include Lobo in the environment. Lobo might itself be implemented as a component, and provide an interface where other components download programs to be executed by Lobo. This way, components and application writers can specify their security requirements in the form of a program, rather than as a deployment descriptor. In particular, the protocol can be supplied or changed after the application itself has been deployed. This makes Obol suitable for use in reflective middleware [7].

Other implementations are also possible: Lobo could be part of the container implementation, or be offered as an external service accessible through the container. When a component is running in a container, the container is part of the trusted computing base (TCB) [44]. Hence, if Lobo is implemented by the container itself, the necessary secure channel between the client and Lobo can be supplied by the container.

Obol and smartcards

Another setting where Obol and Lobo can be applied is systems that use equipment with limited processing, storage, and communication facilities, such as smartcards. When smartcards are used in hostile environments for other purposes than authentication, they need to communicate securely with a system that is trusted by the cardholder [80]. Here the problem stems 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 the client and Lobo 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 Lobo 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 Lobo; Lobo executes the program, and sends the results back to the smartcard.

Obol and certificates

A third setting is one where a service can use existing infrastructure for authentication like an X.509 or SPKI-based PKI to distribute certificates. The server can embed an Obol program in the certificates, and when executed, the program realizes the client side of the protocol over which the service is provided. The client would then obtain the certificates, verify the correctness of the certificate 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 make this desirable.

Weaknesses and future work

Prototype

The current implementation of Lobo is a prototype meant to investigate the basic ideas. We believe the result is encouraging and merit further investigation. But the runtime has not been tested in a real life setting with industrial strength applications and protocols (and was not meant for that either): we believe such tests would raise issues concerning scalability and performance, proper error handling, the application-runtime interface and likely many others. The language itself should be tested with real life applications to see if it's practical: no doubt new operators will be added and old ones modified.

SSH

The SSH protocol described in chapter 2 is a typical example of an industrial strength protocol designed for real life applications. SSH is far more complex than protocols we have implemented so far in Obol. The complexity stems from several factors: There is a lot of state in the protocol compared to a bare bones Needham-Schroeder implementation. For instance, some messages will not be legal before the user has been authenticated, and a rekey operation must not be started if there is already one going on. Another complication factor is that SSH is really a whole architecture composed of multiple protocols working on several levels. For instance, when running the SSH Transport Protocol, most of the messages will be application data belonging to one of the open channels, but there may also be error messages related to failures in the SSH Transport Protocol or maybe rekey requests. A third complicating factor is that there are states which have more than one legal next state. This complicates receiving messages. As a whole, SSH is a rather complex protocol with many purposes. Obol in its current form is no intended for such protocols. Rather it is targeted at purely security related functionality, especially authentication protocols. However, we believe it would be possible to use Obol when implementing an application that provides the same functionality as say a SSH client.

If a SSH client was implemented in Python the Python program would start by loading an Obol script into Lobo, the script would connect to the SSH server and sent the protocol identification message and the algorithm negotiation message. The Obol script would parse the server's reply to the algorithm negotiation message and check if the Lobo runtime could offer some of the algorithms. Alternatively, this could be done by the application. Next, the Obol script would continue by running the key exchange protocol. This is a typical Obol job, and offer no problems. It would generate the session id, and the keys for confidentiality and integrity and be ready to operate in a similar manner to the SSH Transport Layer Protocol. The script could return these values to the application, which in turn could forward it to two new Obol scripts, a send script for sending data to the SSH server and receive script for receiving data. These scripts would be simple loops. In the receive script, every incoming message would be decrypted and the message authentication code (MAC) used in SSH verified. The cleartext would then be returned to the application and processed further in Python. When the application wanted to send something over the secure channel, it would deliver it to the send script and Obol code would encrypt it, add the MAC and send

it to the SSH server. Special control messages, errors, multiplexing channels as in the SSH Connection Protocol etc would be handled by the Python application using the benefits of a full blown programming language.

The SSH client outlined above could not be implemented with the current version of Obol and Lobo. There is for instance no way to receive only the first bytes from a socket; we need that because the length of the packets in the SSH Transport Protocol is encrypted, and we need that length to know how long the packet is and where the MAC starts. However, we believe this and other limitations are a matter of implementation. The real question is if all the switching between Obol and another programming language is worth the added complexity and if the performance penalty from switching between the application and Lobo would be acceptable. This is future work if the overall idea of using a separate language for implementing security is deemed viable.

From Obol to BAN

Obol has been designed to mimic the representation used to describe protocols. The standard protocol description can be viewed as a fixed point on an axis towards a low level implementation in one direction and towards a high level idealization and formal verification in the other (figure 2). We firmly belief Obol is significantly closer to the description than

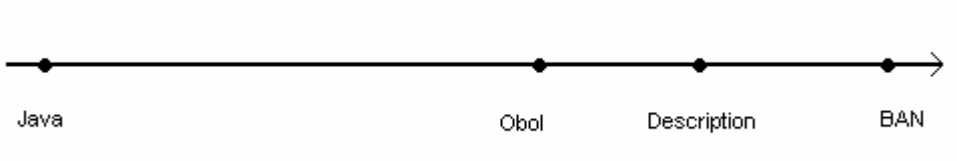

Figure 2 The semantic distance between protocol description, its implementation and formal verification. The axis symbolizes different abstractions and the distance between them.

an implementation in a low level language like C or Java and this is a good thing; the semantic distance should be as short a possible to reduce the number of errors when we go from one abstraction to another. In line with this, future work should strive to close the distance between Obol and protocol descriptions while keeping Obol executable. Having an implementation that matches the description is obviously important, but we still need formal verification techniques like BAN analysis to convince ourselves that the protocol achieves its goals. But the distance between the description and the idealization needed to apply BAN is a problem, just as it is when we go from description to implementation. What we would really like to verify is the implementation, that is, the Obol code. Could it be possible to go directly from Obol to BAN? We don't know, but think it could be worth looking at.

Parsing and protocol verification

The parser is currently very simply and does little beyond matching left and right parenthesis. For instance, it will not detect an invalid operator so `(foo bar)` will be accepted. This is convenient when experimenting with the language design, but a weakness if the implementation where to be used in a real life setting; in that case we would like a parser the user could use to test the Obol program before he executed it. This parser could be extended with heuristics for good protocol design following the line of [3]. For instance, it could give a warning if we tried to sign encrypted data or if we used too short keys. Some of the other principles may be more difficult to add to the parser though; principle 6 states:

> "Be clear what properties you are assuming about nonces. What may do for ensuring temporal succession may not do for ensuring association – and perhaps association is best established by other means."

It's not obvious to us how this could be captured in an algorithm.

Another possible extension to the runtime would be to add a protocol verification tool like the one in [17].

Exception handling

The current exception handling is fairly simple and basically leaves it to the application to decide what to do when an error occurs. The only options offered by the runtime is to abort the script (do nothing) or continue and try the expression that led to the exception again. As we gain experience with the kind of errors that occur in Obol and the way applications use Obol programs, we might be able to see patterns that can be abstracted into an explicit error handling mechanism offered by Obol and the runtime. Indeed, one could envision much more sophisticated handlers [64]:

> "Note, too, that in some possible future world, knowledge representation may have advanced enough that handlers could, rather than act unconditionally on behalf of the signaller, merely return a representation of a set of potential actions accompanied by descriptive information representing motivations, consequences, and even qualitative representations of the goodness of each. Such information might be combined with, compared to, or confirmed by recommendations from other sources of wisdom in order to produce a better result."

The runtime could for instance contain a miniature expert system [68] that decided what to do based on the gravity of the situation.

Tight coupling with the cryptographic library

There is a very tight coupling between the cryptographic library used to implement operators like encrypt/decrypt and sign/verify and the implementation of the runtime. This

means for instance that if a new algorithm is added to the library, it won't be readily available in the runtime. In the future, this could possibly be avoided by adding a layer of abstraction in the runtime so that every hard coded reference to a specific algorithm could be replaced by a query to the cryptographic library to see if the algorithm is available.

Binary compability with other protocol implementations

Our implementation uses an internal message format so if we implemented SSH we would not be able to speak with existing SSH implementations. As explained in chapter 3 and 4, binary compability was not a goal for us, but we think it would be possible to add a way to specify general message formats in future implementations.

Performance

Performance has not been a goal in our work. 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 reason: Security protocols are typically short with few messages and most of the time will be spent either waiting for the network or doing low level cryptographic operations. We can do nothing with the network latency and low level crypto can be handled by hardware accelerators or routines written in C. Therefore, the time spent in the interpreter will in accordance with Amdahl's law [93] be of little importance.

Related work

Other projects related to Obol can be divided into three categories: Formal verification, implementation languages and middleware solutions.

Formal verification

Various methods can be used to analyze protocols to determine whether they achieve their goals, in the hope that the designers can convince themselves and others that the protocol's assumptions, state transitions and desirable goals are sound and attainable [48]. 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. There are several classes of tools available for such analyses [69], for example modal logics [21] and state attainability deducers [48] . 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 they operate on a description of protocols that is not executable like Obol.

Implementation languages

There exist several protocol implementation languages, but none of them are for implementing security protocols. Prolac is one example [41]. 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 implementing security protocols. As mentioned, Prolac compiles to C and the idea is that the resulting code can be embedded in existing code. 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. Another example in the same vein as Prolac is [59].

Middleware

Da Capo++ is a middleware system where many of the application’s needs can be specified in terms of Quality of Service (QoS) values [83]. There is an API to enable applications to compose the system to its needs. Da CaPo++ also places security under the same QoS regime as other resources available to the system; the “degree of privacy” can be specified as a QoS value. The system maintains a database of algorithms, and by using the infrastructure provided by the system, a suitable protocol can be constructed. 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 application. But unlike Obol, there is no way to change the complete security protocol, only the applied algorithms can be changed.

Da CaPo++ and Obol are 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 (as long as the two sides understand each others message formats).

Conclusion

We have designed and implemented Obol, a special purpose high level language for implementing security protocols. Obol has high level abstractions aimed at making the transition from conventional protocol descriptions to running code easy. The Obol runtime, named Lobo, has been implemented and runs security protocols in the form of Obol programs.

References

1. Abadi, M., *Security protocols and their properties*. in *Proceedings of Secure Computation. 27 July 8 Aug. 1999 Marktoberdorf, Germany*, F.L. Bauer and R. Steinbruggen, Editors. 2000. IOS Press, Amsterdam, Netherlands.
2. Abadi, M., et al., *Authentication and delegation with smart-cards.* Science of Computer Programming, 1993. **21**(2): p. 93-113.
3. Abadi, M. and R. Needham, *Prudent engineering practice for cryptographic protocols*. IEEE Transactions on Software Engineering, 1996. **22**(1): p. 6-15.
4. Abelson, H. and G.J. Sussman, *Structure and Interpretation of Computer Programs*. 1996: The MIT Press.
5. Aho, A.V., R. Sethi, and J.D. Ullman, *Compilers. Principles, Techniques, and Tools*. 1986: Addison-Wesley.
6. Andersen, A., *OOPP, A Reflective Middleware Platform including Quality of Service Management*. Dr.Scient Thesis, Department of Computer Service, University of Tromsø. 2002.
7. Andersen, A., et al., *Security and Middleware*. in *The Eight IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003).* 2003. Guadalajara, Mexico.
8. Anderson, A., et al., *Artic Beans: Configurable and Re-configurable Enterprise Component Architectures.* IEEE Distributed Systems Online, 2001. **2**(7).
9. Anderson, R., *Security Engineering*. 2001: Wiley.
10. Anderson, R. and R. Needham. *Robustness principles for public key protocols*. in *International Conference on Advances in Cryptology (CRYPTO 95)*. 1995. Santa Barbara, California, USA. Lecture Notes in Computer Science, **963**.
11. Anderson, R. and R. Needham, *Programming Satan's Computer*, in *Computer Science Today - Recent Trends and Developments*, J. van Leeuwen, Editor. 1996, Springer-Verlag. p. 426-440.
12. Anderson, R.J., *Why cryptosystems fail.* Communications of the ACM, 1994. **37**(11): p. 32-40.
13. Andrews, G.R., *Foundations of Multithreaded, Parallel and Distributed Programming.* 2000: Addison-Wesley.
14. Armour, P.G., *The Five Orders of Ignorance.* Communications of the ACM, 2000. **43**(10).
15. Atkins, D., W. Stallings, and P. Zimmermann, *PGP Message Exchange Formats*, in *RFC 1991*. 1996.
16. Backus, J., *Can Programming be liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*, in *ACM Turing Award Lectures - The First Twenty Years 1966-1985*, R.L. Ashenhurst and S. Graham, Editors. 1977, ACM Press.
17. Basin, D., S. Mödersheim, and L. Vigano. *An On-The-Fly Model-Checker for Security Protocol Analysis*. in *8th European Symposium on Research in Computer Security (ESORICS)*. 2003. Gjøvik, Norway. Lecture Notes in Computer Science, **2808**: Springer Verlag.

18. Bellovin, S.M. 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.
19. Borisov, N., I. Goldberg, and D. Wagner. *Intercepting Mobile Communications: The Insecurity of 802.11*. in *7th Annual International Conference on Mobile Computing And Networking*. 2001. Rome, Italy.
20. Burrows, M., M. Abadi, and R. Needham. *The scope of a logic of authentication*. in *Distributed Computing and Cryptography, DIMACS Workshop*. 1989.
21. Burrows, M., M. Abadi, and R. Needham, *A Logic of Authentication.* ACM Transactions on Computer Systems, 1990. **8**(1): p. 18-36.
22. Clark, J. and J. Jacob. *A survey of authentication protocol litterature*. http://www-users.cs.york.ac.uk/~jac/papers/drareview.ps.gz. 1997.
23. Denning, D.E. and G.M. Sacco, *Timestamps in key distribution protocols.* Communications of the ACM, 1981. **24**(8): p. 533-536.
24. Denning, P.J., et al., *Computing as as Discipline.* Communications of the ACM, 1989. **32**(1).
25. Dierks, T. and C. Allen. *The TLS Protocol Version 1.0*. RFC 2246. 1999.
26. Diffie, W. and M.E. Hellman, *New directions in Cryptography.* IEEE Transactions on Information Theory, 1976. **22**(6): p. 644-654.
27. Ellison, C., et al., *SPKI Certificate Theory*, in *RFC 2693*. 1999.
28. Ferguson, N. and B. Schneier, *Practical Cryptography*. 2003: John Wiley & Sons Inc.
29. Friedman, D.P., M. Wand, and C.T. Haynes, *Essentials of Programming Languages 2nd ed*. 2001: MIT Press.
30. Goldberg, I. and D. Wagner, *Randomness and the Netscape browser*. Dr.Dobbs Journal, 1996: p. 66-70.
31. Graham, P., *ANSI Common Lisp*. 1996: Prentice Hall.
32. Gutmann, P., *Lessons Learned in Implementing and Deploying Crypto Software*. in *11th USENIX Security Symposium*. 2002. San Francisco USA.
33. Hoare, C.A.R. p. Hoare's Dictum: "Premature optimization is the root of all evil in programming". Quoted by Donald E. Knuth in Knuth, D.E. 1989, "The Errors of TeX" in Software - Practice & Experience vol 19(7) pp 607-685, reprinted in "Knuth, D.E. 1992. Literate Programming , CLSI Lecture Notes Number 27, Center for Study of Language and Information, ISBN 0-937073-80-6, where the quote appears on page 276 as follows: "But I also knew, and forgot, Hoare's dictum that premature optimization is the root of all evil in programming". This quote is often attributed to Knuth, but that seems to be a mistake.
34. IBM. *Jikes Compiler Project*. http://www.research.ibm.com/jikes/. 2004.
35. Johnson, S.C. and M.E. Lesk, *Language development tools.* Bell System Technical Journal, 1987. **56**(6): p. 2155-2176.
36. Kaliski, B., *PKCS#5: Password-Based Cryptography Specification v.2.0*, in *RFC 2898*. 2000.
37. Kaliski, B. and J. Staddon, *PKCS #1: RSA Cryptography Specifications v.2.0*, in *RFC 2437*. 1998.
38. Kaufman, C., R. Perlman, and M. Speciner, *Network Security - Private Communications in a Public World*. 2002: Prentice Hall.

39. Kelsey, R., W. Clinger, and J. Rees, *Revised Report[5] on the Algorithmic Language Scheme.* Higher-Order and Symbolic Computation, 1998. **11**(1): p. 7-104.
40. Khol, J. and C. Neuman, *The Kerberos Network Authentication Service (V5),* in *RFC 1510*. 1993.
41. Kohler, E., M.F. Kaashoek, and D.R. Montgomery, *A readable TCP in the Prolac protocol language.* ACM SIGCOMM, 1999: p. 3-13.
42. Kon, F., et al., *The Case for Reflective Middleware.* Communications of the ACM, 2002. **45**(6): p. 33-38.
43. Lai, X., *On the Design and Security of Block Ciphers*, in *ETH Series in Information Processing*. 1992, Hartung-Gorre Verlag: Konstanz, Switzerland.
44. Lampson, B., et al., *Authentication in distributed systems: theory and practice.* ACM Transactions on Computer Systems, 1992. **10**(4): p. 265-310.
45. Landin, P.J., *The next 700 programming languages.* Communications of the ACM, 1966. **9**(3): p. 157-166.
46. MacLennan, B.J., *Principles of Programming Languages*. 1999: Oxford University Press.
47. McConnell, S., *Code Complete: A practical Handbook of Software Construction, 2nd Ed.* 2004: Microsoft Press.
48. Meadows, C., *Formal Verification of Cryptographic Protocols: A Survey*, in *Advances in Cryptology - Asiacrypt '9*. 1995, Springer-Verlag. p. 133-150.
49. Menezes, A.J., P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*. 1997: CRC Press.
50. Microsoft. *Windows 2000 Kerberos Authentication.* White paper. http://www.microsoft.com/windows2000/docs/kerberos.doc. 1999.
51. Mitchell, J.C., *Concepts in programming languages*. 2003: Cambrigde University Press.
52. Myrvang, P.H., *Persoval communication on upgrading OpenSSL in Gentoo*. 2003.
53. Needham, R., *Names*, in *Distributed Systems*, S. Mullender, Editor. 1993, Addison-Wesley & ACM Press.
54. Needham, R. and M.D. Schroeder, *Using Encryption for Authentication in Large Networks of Computers.* Communications of the ACM, 1978. **21**(12): p. 993-999.
55. NIST, *186-2 Digital Signature Standard*, in *FIPS PUB*. 2000.
56. NIST, *197 Advanced Encryption Standard (AES)*, in *FIPS PUB*. 2001.
57. NIST, *180-2 Secure Hash Standard*, in *FIPS BUB*. 2002.
58. Norvig, P., *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. 1992: Morgan Kaufman.
59. Patel, P., et al., *Upgrading Transport Protocols using Untrusted Mobile Code*. in *Symposium on Operating System Principles (SOSP)*. 2003. New York, USA.
60. Paulson, L.C., *Mechanized Proofs for a Recursive Authentication Protocol*. in *10th Computer Security Foundations Workshop*. 1997. IEEE Computer Society Press.
61. Perlis, A.J., *Epigrams in Programming.* ACM SIGPLAN Notices, 1982. **17**(9): p. 7-13.
62. Pfitzmann, B., M. Schunter, and M. Waidner. *How to Break Another Provable Secure Payment System*. in *EUROCRYPT '95*, L.C. Guillou and J.-J. Quisquater, Editors. 1995. St.Malo, France. Lecture Notes in Computer Science, **921**: Spring-Verlag, NY.

63. Pitman, K.M., *Exceptional situations in Lisp*. in *First European Conference on the Practical Application of Lisp (EUROPAL '90)*. 1990. Churchill College, Cambrigde, UK.
64. Pitman, K.M., *Condition Handling in the Lisp Language Family*, in *Advances in Exception Handling*, A. Romanovsky, et al., Editors. 2002, Springer.
65. Queinnec, C., *Lisp in Small Pieces*. 1996: Cambrigde University Press.
66. Ramming, J.C., *Proceedings of the USENIX Conference on Domain-Specific Languages*. in *USENIX Conference on Domain-Specific Languages*, J.C. Ramming, Editor. 1997. Santa Monica, California, USA.
67. Rivest, R., *The MD5 Message Digest Algorithm*, in *RFC 1321*. 1992.
68. Russell, S.J. and P. Norvig, *Artificial Intelligence: A Modern Approach*. 2002: Prentice Halls.
69. Ryan, P. and S. Schneider, *Modelling and analysis of security protocols*. 2001: Addison-Wesley.
70. Ryan, P.Y.A. and S.A. Schneider, *An attack on a recursive authentication protocol. A cautionary tale.* Information Processing Letters, 1998. **65**(1): p. 7-10.
71. Samar, V. and R.J. Schemers, *Unified Login with pluggable authentication modules (PAM)*, in *Open Software Foundation RFC 86.0*. 1995.
72. Schneier, B., *Applied Cryptography*. 1996: Wiley.
73. Sebesta, R.W., *Concepts of Programming Languages*. 2002: Addison-Wesley.
74. Skogan, T., *Common Lisp Cryptography (CLC) - A cryptographic library in Common Lisp*. INF 3982 Project Report, Department of Computer Science, University of Tromsø. 2003.
75. Smith, B., *Reflection and Semantics in a Procedural Language*. Ph.D. thesis, MIT Laboratory for Computer Science, MIT. 1982.
76. Spinellis, D., *Notable design patterns for domain-specific languages.* The Journal of Systems and Software, 2001. **56**(1): p. 91-99.
77. Spinellis, D. and V. Guruprasad. *Lightweight languages as software engineering tools*. in *Proceedings of the USENIX Conference on Domain-Specific Languages*, J.C. Ramming, Editor. 1997. Santa Monica, California, USA.
78. Stabell Kulø, T., *Private Computing - The Trusted Digital Assistant*. Ph.D., Computer Science Department, University of Tromsø. 2002.
79. Stabell Kulø, T., *Personal communication on the size of various programming tools.* 2004.
80. Stabell Kulø, T., R. Arild, and P.H. Myrvang. *Providing authentication to messages signed with a smart card in hostile environments*. in *USENIX Workshop on Smartcard Technology*. 1999. Chicago, Illinois, USA.
81. Stallings, W., *Cryptography and Network Security - Principles and Practice*. 1999: Prentice Halls.
82. Steele, G.L., Jr., *Common Lisp: the Language 2ed*. 1990: Digital Press.
83. Stiller, B., et al., *A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience*. IEEe JSAC: Special Issue on Middleware, 1999. **17**(9): p. 1614-1631.
84. Stubblefield, A., J. Ioannidis, and A.D. Rubin, *Using the Fluhrer, Mantin and Shamir attack to break WEP.*, in *Technical Repport TD-4ZCPZZ, revision 2*. 2001, AT&T Laboratories.

85. Syverson, *Knowledge, belief and semantics in the analysis of cryptographic protocols*. Journal of Computer Security, 1992. **1**(3): p. 317-334.
86. Syverson, P. and I. Cervesato. *The Logic of Authentication Protocols*. in *Proceedings of Foundations of Security Analysis and Design*, R. Focardi and R. Gorrieri, Editors. 2001. Springer.
87. Tanenbaum, A.S., *Modern Operating Systems*. 2 ed. 2001: Prentice Hall.
88. Tanenbaum, A.S. and M. van Steen, *Distributed Systems - Principles and Paradigms*. 2002: Prentice Hall.
89. Thompson, K., *Reflections on Trusting Trust*, in *ACM Turing Award Lectures - The First Twenty years 1966-1985*, R.L. Ashenhurst and S. Graham, Editors. 1983, ACM Press.
90. Turing, A.M., *On computable numbers, with an application to the entscheidungsproblem.* Proceedings of the London Mathematical Society, 1936. **42 (1936-37)**: p. 230-265.
91. Viega, J. and G. McGraw, *Building Secure Software*. 2002: Addison-Wesley.
92. Viega, J., M. Messier, and P. Chandra, *Network Security with OpenSSL*. 2002: O'Reilly.
93. Wilkinson, B. and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. 1998: Prentice Hall.
94. Ylonen, T., et al. *SSH Connection Protocol draft-ietf-secsh-connect-17.txt*. IETF Network Working Group, Internet-Draft. 2003.
95. Ylonen, T., et al. *SSH Authentication Protocol draft-ietf-secsh-userauth-17.txt*. IETF Network Working Group, Internet-Drafts. 2002.
96. Ylonen, T., et al. *SSH Transport Layer Protocol draft-ietf-secsh-transport-16.txt*. IETF Network Graoup, Internet Draft July 14 2003. 2003.
97. Ylonen, T., et al. *SSH Protocol Architecture draft-ietf-secsh-architecture-14.txt*. IETF Network Working Group, Internet-Draft, July 14 2003. http://www.ietf.org. 2003.
98. Zimmermann, P., *PGP source code and internals*. 1995: MIT Press.