Gaining Flexibility by Security Protocol Transfer

Per Harald Myrvang*
Bodø Graduate School of Business, Norway
permyr@gmail.com

Tage Stabell-Kulø
University of Tromsø, Norway
task@cs.uit.no

Abstract

Even though PDAs *in general—but smartcards in particular—can be trusted to keep secrets, because they have meager resources, including them in security protocols is difficult. The* PDAs*—and again smartcards in particular—ends up as a mere key-store and their processing power is not used.*

We describe a mechanism that allows such anemic computers to fully participate in protocols, even if the protocol in question by far exceeds their capabilities. This is done by means of machinery for transferring, at runtime, the protocol proper to a more powerful machine.

We describe the mechanisms that makes this possible (mainly a domain specific programming language named Obol [5] and its implementation) and we discuss the credentials and certificates needed for the solution to maintain correctness.

1. Introduction

Of all the security protocols in use, some (for example SSL/TLS, Kerberos, S/MIME and SSH) provide support for PDAs and smartcards, typically for storing keys, or performing initial key setup. Compared to the security protocol proper, the PDA's rôle is highly specialized, and not very exciting. In other words, only the ability to keep secrets is utilized, not it's processing capability The reason is simple: most security protocols are simply too large.

When the card holds private keys, it is for all intent and purposes part of the Trusted Computing Base (TCB [3]) and its potential as contributing to computations within the TCB far exceeds its processing power alone. The TCB is the set of machines (and their software) that are trusted. These "islands" are connected by the use of cryptography and maybe the biggest challenge in security is to minimize the TCB while maximizing the ability to adopt to changes. It is in this context that the trust placed on PDA is so valuable.

We will discuss mechanisms that makes it possible for the processing capability be fully exploited so that the PDA's part in protocols go beyond the trivial.

At the core of our design is the ability for a PDA, or any resource deprived computer, such as a smartcard, to transfer to another (more powerful) computer a security protocol for execution in a seamless manner. The scenario is one where a user wants to access a service with his PDA (i.e. smartcard). He is informed by the PDA that it is unable to execute the protocol because it is too large (or computationally overwhelming), so he instructs the PDA to transfer the protocol to a more powerful computer. Access to the service is resumed when the remote computer signals that the protocol did run to completion. For this to be possible in a general manner it must be possible to transfer the protocol.

For simplicity we will assume that this other computer is part of the user's TCB; possibly residing safely at home. As will be evident, more complex settings can be constructed by means of elaborate certificates.

This is achieved by a high-level security protocol programming language; protocols are transferred by means of a script that can be created on-the-fly even on a smartcard. By using this mechanism, and carefully crafted certificates, protocols can be initialized at one point, transferred, executed to completion somewhere else, and the result then transfered back to the initiator.

If the PDA had sufficient resources, a jar of pre-compiled Java byte-code could have been constructed, signed and transfered. However, for a non-trivial protocol this is not feasible with contemporary technology. We provide a solution to this problem.

The rest of the paper is structured as follows: In Section 2 we present one of the important enabling technologies for our idea, the domain-specific programming language Obol; In Section 3 we describe the mechanism in detail, the logic of authentication used for analysing it, and several examples; Some related work is briefly described in Section 4; Section 5 concludes.

2. The Obol Security Protocol Language

Our system hinges on the ability to express non-trivial security protocols in a manner so dense that even a smart-

card can deal with them. Obol is a high-level, domain-specific security protocol programming language that we have developed and implemented for this purpose [4, 5]. The language is only concerned with issues deemed to be relevant to the execution of security protocols. The issues not directly relevant are handled by the language's runtime. Unlike many security specification languages and analysis tools, Obol deliberately makes no global assumptions about the security protocol it implements. In particular, there are no assumption that one single authority control all participants – meaning that an Obol program implements *one* side of the protocol. As consequence there is no requirement that all participants use Obol, as long as non-Obol users can send messages that are parseable by those that do use Obol: i.e. that the message transport format is known.

Protocols implemented in Obol are called *scripts*. Obol scripts are short, textual programs that can be read and understood by people. An executable implementation language must express clearly what's going on, and cannot afford the ambiguity of more abstract notations, e.g. the traditional "Alice-and-Bob" notation. In this case, Obol scripts are more verbose than the abstract notation, but significantly shorter than the same protocol implemented in a low-level language such as C or Java. For example, Message 5 from [6] in "Alice-and-Bob" notation:

$$O \rightarrow S: \quad \{C, X, H(X, H(Y))\}_{K_O^{-1}}$$

Given the high abstraction of the above message, there are many possible implementation. For example, a code fragment implementing O could be:

```
(believe K_O (load "O-keyfile")
             keypair ((alg RSA)))
(believe H_XY (sign "SHA1" X
                    (sign "SHA1" Y)))
(send S  C X H_XY (sign (private-key K_O)
                        C X H_XY))
```

The corresponding code fragment for S would be:

```
(believe K_O (load "O-pubkey")
             public-key ((alg RSA))
(receive O  C X *H_XY
            (verify K_O C X *H_XY))
```

In the high-level notation, keys are simply correctly placed and typed symbols, while in Obol key management is explicitly exposed. Furthermore, one must explicitly state what is signed[1] to facilitate signatures on computed data that is not actually transmitted. Also, notice that the above Obol code is directly executable by the Obol runtime *as is*—no further compilation or other transformation are required (apart from the initialization of O and S, not shown for brevity).

As the above example shows, Obol is more verbose than the abstract "Alice-and-Bob" notation, but the semantics are clearer — there is no ambiguity regarding what the message actually consist of, nor what preparations are required for building and sending the message. This in contrast to the "Alice-and-Bob" notation, where the message could be interpreted as the transfer of just the signature, or any combination of the message components and the signature.

The Obol runtime is designed as a stand-alone service and is implemented in Java. One of the motivations for Obol was to reduce the entanglement between the application, protocol machinery, and security policy implementations. It is this separation of concerns that enables us to transfer a protocol for execution remotely.

3. The Mechanism

We wish to use the flexibility provided by Obol to enhance and extend the ability of PDAs and smartcards to participate in (security) protocols. In essence, we use Obol protocol scripts both as description and context in which a given protocol is to be evaluated. Also, the use of Obol allows for a flexible "bootstrapping," as described in Section 3.3. By analysing the authentication statements using the logic of [3], we have some confidence in the soundness of our reasoning.

Taken together, this allows us to dynamically delegate the part of the PDA to a more powerful remote computer within the TCB. If care is taken into designing the protocols, we believe this increased flexibility can be gained without sacrificing security. As with all delegation, this requires that the right assumptions are made, and the appropriate credentials to be generated and presented when needed.

The main idea is to move away from the usual PDA authentication form, typically expressed (in the logic of [3]) as

$$\textit{PDA}\ \textbf{says}\ x \quad \text{or} \quad (\textit{PDA} \mid \textit{User})\ \textbf{says}\ x$$

meaning that the PDA merely identifies itself (speaks for itself), or that it clearly identifies whom it speaks on behalf of, and can show appropriate credentials to this effect. This in contrast to our goals, which form more complex expressions such as

$$(\textit{Machine}\ \textbf{for}\ \textit{PDA})\ \textbf{as}\ \textit{Script}\ \textbf{says}\ x$$

which can be interpreted as "The computer *Machine* acts on behalf of *PDA* while running protocol instance *Script*." Such a statement resolves to an equivalency between the

[1] The Obol operators sign and verify have three modes, determined by the *key* argument type: asymmetric (i.e. public-key based "digital signatures"), symmetric (i.e. MACs), and hash (i.e. message digest algorithm name).

PDA and the machine running the protocol, when the machine says it acts on behalf of the PDA, and the PDA confirms it. The machine *is* the PDA, for all authentication purposes (obviously there must be some safeguards, such as an expiration time, or remote attestation). The expression is parameterized not only by the conveyed meaning x, but by means of the script *how* the meaning x is conveyed.

By adding one level of indirection both on protocol execution location (introduced by the *Machine* principal), *and* on the protocol content (by the *Script* rôle), we believe greater flexibility has been obtained.

3.1. Logic of Authentication

To analyze and reason about entities of authority (i.e. principals), authentication, delegation and so on, we use a logic for authentication from [3]. Without going too far into excruciating details, this logic deals with principals (entities that can say something), and how these can make statements about other principals. The logic is amazingly information-dense, and a presentation doing it justice far exceeds both the scope and space of this paper.

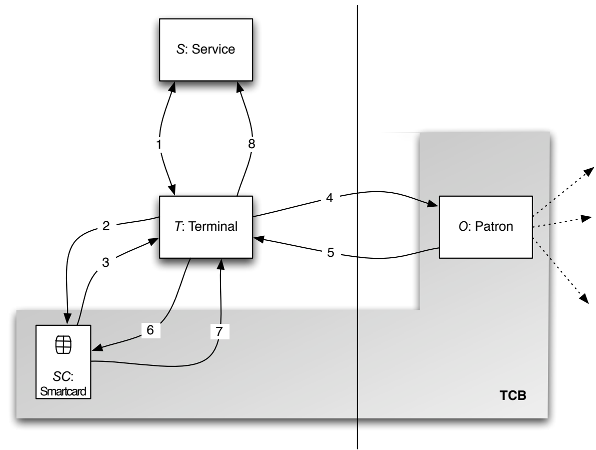
We have found it useful to make use of an extension, a new principal construct $A\backslash P$, meaning A executing program P. This is a short-hand for the existing rôle construct, and we use it solely to reduce verbosity. The statement $A\backslash P$ **says** x is derived from A **as** P **says** x. By rule (R1)[3] this is the same as $A|P$ **says** x. The statements needed to obtain this delegation is:

- A delegates to B by generating A **says** $B|A \Rightarrow B$ **for** A.
- B acknowledges the delegation by issuing the receipt $B|A$ **says** $B|A \Rightarrow B$ **for** A.
- From this it follows that $B|A \Rightarrow B$ **for** A.
- After or during execution of a program P, B produces the statement $B|A$ **as** P **says** x, where x is the output or result of the program.

Anybody in possession of the above statements can derive (B **for** A) **as** P **says** x. We compress the above into the notation $A\backslash P$, to be read as "A executing program P", where B is unimportant except as an enabling credential to reach this conclusion – remember, B has entered the rôle of A in context of P, and is no longer acting on behalf of itself. Observe that a statement involving the program directly, such as A **says** $P \Rightarrow A$, is invalid, since only a principal can speak on behalf of other principals. P is a program, i.e. a specification, and cannot *do* anything by itself. Any principal executing P, however, acts according to P's specification, or in P rôle.

Assuming a rôle should not be taken lightly, however. We assume that B is unwilling to execute arbitrarily received programs, so B must somehow understand that it is A sending the program P, which is safe for B to execute. In practice, we get A to sign P, which to the logic is a statement saying A **says** $P \in$ (Programs $\in$ TCB).

The logic of authentication allows us to express these concerns at a high level of abstraction, and makes apparent the pre-requisites for obtaining safe and sound delegation.



(a) Overview

N^o	**Meaning**
1	Initial contact between S, T and SC
2	T **says** Request-protocol(P)
3	$\{P_{obol}\}_{K_{SC}^{-1}}$, SC **says** $O\|SC \Rightarrow O$ **for** SC. Optionally, $O\|SC$ **says** $O\|SC \Rightarrow O$ **for** SC.
4	Forwarded Step 3, session/service parameters
5	$A\backslash P_{obol}$ **says** x, i.e. P_{obol}-dependent reply from O, including $O\|SC$ **says** $O\|SC \Rightarrow O$ **for** SC, if not sent in Step 3.
6	Forwarded Step 5 to SC
7	Protocol P_{obol}-dependent response to Step 5
8	Closing contact between S and T.

(b) Steps

Figure 1: Partial remote protocol execution. Depending on the circumstances, there might be multiple rounds before the session is finished.

3.2. Example: Partial Remote Protocol Execution

The goal of this example scenario is to use a resource deprived PDA in a protocol scenario beyond its capabilities, while still having it participate. This is done by having the PDA hand over parts of the protocol session to a trusted machine with more computational abilities – the patron. The PDA is still part of the protocol, and can perform important functions such as signing, key exchange, en-/decryption, verification and so on.

The mechanism works as follows (see Figure 1): Upon using a service S by means of some protocol P, the terminal

T queries the PDA SC if it wishes to use this protocol. If the protocol is known but beyond the PDA's ability to execute on-board, SC prepares a description of the protocol: the Obol script P_{obol}. The PDA can adjust P_{obol} to fit the current situation, by e.g. filling in S and T's identities and/or keys. For the delegation to work, the necessary credentials must also be produced:

$$SC \textbf{ says } O|SC \Rightarrow O \textbf{ for } SC \quad (1)$$

$$O|SC \textbf{ says } O|SC \Rightarrow O \textbf{ for } SC \quad (2)$$

$$SC \textbf{ says } P_{obol} \in \text{TCB} \quad (3)$$

Of these, only credential (3) must be generated "on-site", while the others ((1) and (2)) may be prepared in advance. In fact, SC need not provide credential (2) as it can be provided by O at a later stage.

Continuing with the example, the terminal T is asked to forward P_{obol} to the patron O, as well as any subsequent communication between O and SC. The latter is useful because it allows SC and O to establish timeliness of the credentials and delegation, as well as allowing SC to participate in the protocol itself (exactly how this participation takes form depends on the P_{obol} implementation).

Upon receiving P_{obol} and the above credentials (of which (2) can be retrieved or generated on-the-fly by O), O executes P_{obol} and by doing so generates $A \backslash P_{obol}$ **says** x. Finally, S collects all credentials (forwarded by T), and can derive that SC **says** x. By [3] axioms and theorems (D1), (P10), (P11) and (P1), it can be derived that SC **says** $O|SC$ **says** x, which reduces to SC **says** x, which is what S need to believe.

For this to work, there are some assumptions that must be made. In particular, the scheme requires that S doesn't care whom it services as long as the credentials are valid. Also, S must accept composite credentials, i.e. that the patron is allowed to act together with the PDA, as the PDA. T's part is to interact with the PDA, on behalf of the service and itself, i.e. it authenticates the PDA holder by means of a PIN, biometrics or some other means. In this case, the PDA hands over part of the protocol execution to the home-TCB. The PDA awaits some kind of result from the patron, which triggers the PDA to provide some required response, e.g. a signed receipt, a key, or some other data of value. The portion of protocol executing at the patron can be provided by the PDA. Alternatively, there may be a set of prepared Obol protocols at the patron, requiring only an protocol-ID and parameters to be transferred. Communication between the PDA and the home-TCB must at least be integrity-protected, while the need for secrecy depends on the sensitivity.

This abstract scenario allows the PDA to participate in a protocol that is "larger-than-life" (relative to the PDA), while still keeping its presence crucial. Below we will give a more concrete example of how this scenario may be realized.

```
(script "from smartcard"
       :doc "Executed by patron on behalf of smartcard"
       (believe Ku |OWVmNTAwZmQyZjFhOTU0MQo=|
                   shared-key ((alg AES)))
       (believe A "auth-server.org" host)
       (believe G "tgs.org" host)
       (believe U "smartcard-kerberos-user-id")
       (believe S "10.0.0.1" host)
       [format spki]
       [use "kerberos-v5 Client"
                    :input A G Ku U S
                    :result Kcs]
       (believe Kp-sc (load "shared-smartcard.key")
                      shared-key ((alg AES)))
       (believe *K (encrypt Kp-sc Ksc))
       [returns *K])
```

(a)

```
(script "patron"
       [self "localhost:12345"]
       [format spki]
       (believe Kp-sc (load "shared-smartcard.key")
                      shared-key ((alg AES)))
       (believe Ksc-pub (load "smartcard.pubkey")
                        public-key ((alg RSA)))
       (believe K_O (load "patron.keypair")
                    keypair ((alg RSA)))
       (receive *T (decrypt Ksc-p *script
                            (verify Ksc-pub *script)))
       [use *script :returns x]
       ;; O for SC says x
       (send *T  (public-key K_O) "for" Ksc-pub x
             (sign (private-key K_O)
                   (public-key K_O) "for" Ksc-pub x)))
```

(b)

Figure 2: Example scripts for delegating Kerberos key-exchange to patron: (a) Script from PDA, in effect Message 3 of Figure 1. The symbols A, G, U and S are configured by the PDA. Removing pretty-printing, comments and whitespace, this script is 237 octets long. (b) Script representing patron. The script is simplified, so features like key lookup based on PDA identity is not show.

Example Scripts. Figure 2 shows the (actual) scripts needed for a scenario where a service S expects further interaction with the PDA to happen over an encrypted channel, and that the shared key used for this to be obtained by means of Kerberos. In this case this is beyond the PDA, so it configures an Obol-script (a simple search-and-replace operation), signs it, encrypts it using a key shared between the PDA and the patron, before it asks the terminal to forward a message containing the signed script to the patron. The signed and encrypted script itself forms credential (3). This script, as shown in Figure 2(a), simply instructs the patron what to do – invoke the Kerberos client-side protocol with the given arguments, and encrypt the result before returning it. The credential (1) is encoded within the script, by telling the patron about the Kerberos-domain key Ku.

The patron-side script, show in Figure 2(b), is started in

advance like any server, configures itself by loading keys, then it waits until it receives a message that can be successfully decrypted and verified as originated from the PDA. The patron then executes the received script. Other configurations could want to enforce a policy that required received scripts to be examined first, to verify that they are safe. This would simply involve changing the "`[use...]`" statement to e.g. "`[returns *script][input x]`", which returns the received script to an application, which then verifies and executes it in a separate Obol thread, and returns the result to the patron script. The patron then constructs a message containing one of may possible encodings of the O **for** SC **says** x credential, and sends this back to the terminal, which will forward it to the PDA. The message itself form credential (2). The PDA verifies that x was said by the patron, decrypts x, retrieves the through-Kerberos-exchanged shared key, and can now use that for subsequent interaction with the service. The service, being in contact with the Kerberos infrastructure, knows upon the PDA usage of the shared key, that whom it communicates with has been authenticated to its satisfaction. Notice that the service doesn't actually follow the evidence step-by-step towards SC **says** x, but that we have constructed the messages so that a future audit will make such a conclusion.

3.3. Variations: Hand-off and Parameterization

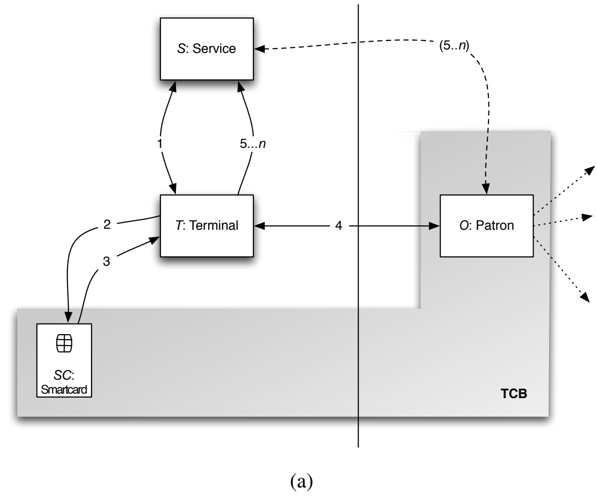
In addition to delegation and partial remote protocol execution, there are two other ways in which Obol enable new functionality.

PDA Rôle Hand-off. A PDA can be hoisted into the domain of too-large protocols by delegating the entire rôle of the PDA to a more powerful machine within the same TCB.

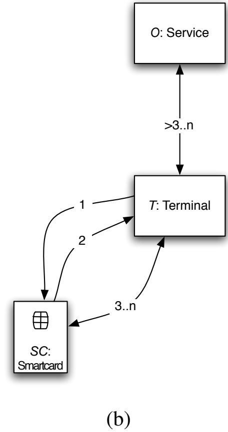
This is a variation of the partial delegation of control presented in Section 3.2. Instead of participating in the protocol after the delegation messages have been sent, the PDA delegates the responsibility of executing the entire protocol to the patron, see Figure 3(a). The degree of delegation may be range from complete (the home-TCB is, after all, trusted), to partial (restricted in authorization, scope, or time-frame). It should be noted that a unrestricted delegation (called a "hand-off" in [3]) to the home-TCB is most likely unwise – the operation is too powerful, and transcends the local scope so completely that it is very difficult to envision (all) the ramifications.

In order to limit the impact of the hand-off it would be safer, security-wise, to create a temporary key-pair K_{temp}, delegate to it, and use that as the PDA's representative. To do this, SC generates

$$SC \textbf{ says } K_{temp}|SC \Rightarrow K_{temp} \textbf{ for } SC \qquad (4)$$



(a)



(b)

Figure 3: Variants of control transfer: (a) Delegation of PDA rôle to O. (b) Parameterized PDA protocol.

and sends (preferably with the private key component encrypted) K_{temp} to O along with (4). After taking control of the key-pair, O generates the delegation receipt using K_{temp}:

$$K_{temp}|SC \textbf{ says } K_{temp}|SC \Rightarrow K_{temp} \textbf{ for } SC \qquad (5)$$

The authentication expression

$$(K_{temp} \textbf{ for } SC)\backslash P_{obol} \textbf{ says } x$$

would then expand to (O still being the patron):

$$(O \textbf{ for } (K_{temp} \textbf{ for } SC)) \textbf{ as } P_{obol} \textbf{ says } x \qquad (6)$$

Note that we get O **for** K_{temp} because it is O that uses K_{temp}. Finally, given the certificates (4), (5), (1), (3), and (2) this resolves to SC **says** x which is what S would want.

After the hand-off, as shown in Figure 3(a), the terminal may function as a proxy between S and O, or S and O may

continue together so that the rest of the protocol executes without T. The PDA is not involved after the hand-off has taken place.

Parameterized Protocol Execution. Some protocols may be within the capabilities of the PDA, and does not require a patron. It is still useful to express the security protocol using Obol, as this provides the PDA peer with an executable specification of it's side of the protocol (this is essentially [4] applied to a PDA scenario).

In this scenario the PDA acts as a medium for the Obol script of its peer, and upon request provides the script (possibly configured for the particular request), see Figure 3(b). The provided script must be authenticated, e.g. by some form of signature, so that the following hold:

$$SC \textbf{ says } SC|P_{obol} \Rightarrow SC \textbf{ as } P_{obol}$$

The meaning is: when the PDA follows the protocol, it acts on behalf as itself, thus forming a link between the PDA and the protocol that convinces the peer to follow the protocol. Note that P_{obol} might be executed either by T or S, depending on what the protocol addresses: communication (T), or a service (S). T or S can inspect the script to convince themselves that the protocol does nothing undesirable.

This scenario can be used to bootstrap the more advanced scenarios involved delegation, as described in the previous section.

4. Related Work

A JINI-based smartcard middleware is presented in [2]. Here, the goal is to dynamically install middleware components in the terminal that export card services. The solution requires access to an on-line repository of components, and does not address the communication between card, terminal and service. Security is explicitly not addressed.

A open protocol for smartcard interoperability is presented in [1]. Here, a low-level middleware is used to connect the card and the service host, using plug-ins for different types of cards. A PKCS#11 module has been developed, allowing integration with with several applications supporting this standard, e.g. Mozilla, and OpenSSH.

Both of these are important for our work in that they address the infrastructure needed to deploy our mechanism.

5. Conclusion

There are several benefits that arise from our mechanism: By embedding protocol descriptions in certificates, protocols stored on the PDA can be updated by downloading an appropriate certificate (which will be verified by thePDA). This is similar to other efforts (e.g. [1, 2]), where signed code is uploaded to a smartcard.

Furthermore, the PDA need not interpret the embedded protocol (i.e. it can run a native-code version), allowing it to selectively present the protocol it will follow to the terminal (or to the service). This can be thought of as the PDA providing the driver-code needed by the terminal to follow the given protocol. It can also adapt (rewrite) the protocol, to fit the circumstances (i.e. parameterization).

Finally, data may be forwarded from the PDA to a patron within the home-TCB, and any results returned to the PDA, thus allowing the PDA to participate. Terminals (or services) may execute Obol protocols against the home-TCB, in partial or complete fulfillment of the purpose of using the PDA. The PDA may also be allowed to delegate its rôle to patrons in the home-TCB. When used in this manner, the PDA can be viewed as a trusted protocol initiator.

There are many advantages to PDAs and smartcards – they are relatively secure, and very easy to use, even if their impact have been limited in scope. We believe that we have described a flexible and powerful mechanism that extends the usability of smartcards, without compromising security.

Acknowledgements

Thanks must go to Andrea Bottoni and Terje Fallmyr for their valuable comments and suggestions.

References

[1] T. Cucinotta, M. D. Natale, and D. Corcoran. A protocol for programmable smart cards. In *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, page 369, Washington, DC, USA, 2003. IEEE Computer Society.

[2] R. Kehr, M. Rohs, and H. Vogt. Mobile code as an enabling technology for service-oriented smartcard middleware. In *DOA '00: Proceedings of the International Symposium on Distributed Objects and Applications*, page 119, Washington, DC, USA, 2000. IEEE Computer Society.

[3] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distribued systems: theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.

[4] P. H. Myrvang. Parameterised communication. *Dr. Dobb's Journal*, 31(10):22–26, Oct. 2006.

[5] P. H. Myrvang and T. Stabell-Kulø. The obol protocol language. In *14th International Workshop on Security Protocols Cambridge, England, March 2006*, (to appear) LNCS. Springer Verlag, 2005.

[6] T. Stabell-Kulø, R. Arild, and P. H. Myrvang. Providing authentication to messages signed with a smart card in hostile environments. In *Proceedings from the USENIX Workshop on Smartcard Technology*, pages 93–99, May 1999.