Wireless Authentication in χ-Spaces

Federico Crazzolara
Giuseppe Milicia

BRICS Report Series RS-03-10

ISSN 0909-0878 February 2003

Wireless authentication in χ-Spaces

Federico Crazzolara[*] Giuseppe Milicia[*]

BRICS, Center of the Danish National Research Foundation
federico@ccrl-nece.de, milicia@brics.dk

Abstract

The χ-Spaces framework [CCM02b] provides a set of tools to support every step of the security protocol's life-cycle. The framework includes a simple, yet powerful programming language which is an implementation of the Security Protocol Language (SPL) [CW01]. SPL is a formal calculus designed to model security protocols and prove interesting properties about them. In this paper we take an authentication protocol suited for low-power wireless devices and derive a χ-Spaces implementation from its SPL model. We study the correctness of the resulting implementation using the underlying SPL semantics of χ-Spaces.

1 Introduction

Security protocols describe a strategy which allows two or more parties to securely exchange information over an untrusted media. Pragmatic experience leads us to believe that the design and implementation of security protocols is a challenging task. Indeed the few engineering guidelines available to security protocol designers offer little, if any, guarantee on the quality of their design efforts. A painful common denominator of a number of published security protocols is that they are, later on, found to be flawed in one way or another. Striking examples of such trend are the well-known Needham-Schröder protocol [NS78], an attack against which was found almost twenty years after its publication in [Low96], but also less over-cited experiences such as the Π protocol of Woo and Lam [WL94]. The list is uncomfortably long [CJ97].

As a reaction to this situation, *formal methods* have been used, with some success, to study newly introduced security protocols. However, a different, and often overlooked, aspect of the field is the danger involved in the step that brings a correct formal specification of a security protocol to its actual real-life implementation. A protocol, which can be easily specified in few lines, results

[*]Chi Spaces Technologies ltd.

in thousands of lines of code. Believing that the security properties of the concise specification will carry forward to its implementation is, more than anything else, an act of faith. This faith is often misplaced. An embarrassing example of such situation, is the implementation of the SSL protocol in the Netscape web browser. The SSL protocol is well-understood and has been proved correct in several studies. Its implementation in the Netscape browser was, however, found to be flawed [ACR00].

The χ-Spaces framework (pronounced *key* spaces), and in particular the χ-Spaces programming language [CCM02b], is meant to fill the gap between a formal specification of a security protocol and its implementation. χ-Spaces provides a concise, yet powerful, scripting language specifically designed for the development of security protocol. This language is an implementation of the *Security Protocol Language* (SPL) [CW01], a calculus for the specification and analysis of security protocols. The χ-Spaces language and SPL are closely related, indeed, the SPL semantics and proof techniques are easily adapted to analyse χ-Spaces programs [CCM02b]. χ-Spaces has been successfully applied to the study and implementation of real-life protocols [CCM02c, CM02].

χ-Spaces is based on the JAVA programming language [GJSB00] and as a consequence enjoys its platform independence. χ-Spaces does not rely on point-to-point communication, a *space*, a remotely available storage area, is used instead. This architecture makes χ-Spaces ideal in a wireless and highly dynamic settings [CCM02a]. Our current research aims at the deployment of efficient protocols for low-power wireless devices such as Personal Digital Assistants (PDAs), mobile phones, etc.

From a practical point of view, χ-Spaces is compatible with the Java 2 micro edition (J2ME) [RTVH01] and PersonalJava frameworks. At present, the J2ME technology has been accepted by a number of companies and J2ME aware devices are on the verge of flooding the market. Top-performance PDAs, such as the Compaq iPAQs, are powerful enough to support the PersonalJava virtual machine.

The design of security protocols for low-power devices introduces a number of new concerns which we discuss. We then introduce the SSMAKEP (server side mutual authentication and key exchange protocol) protocol [WC01b], which we found meeting our criteria. We provide an implementation of the protocol in χ-Spaces, and study its performances and its security properties (using the SPL proof methodology). The Appendix discusses an attack scenario on the SSMAKEP protocol, in the presence of a compromised server.

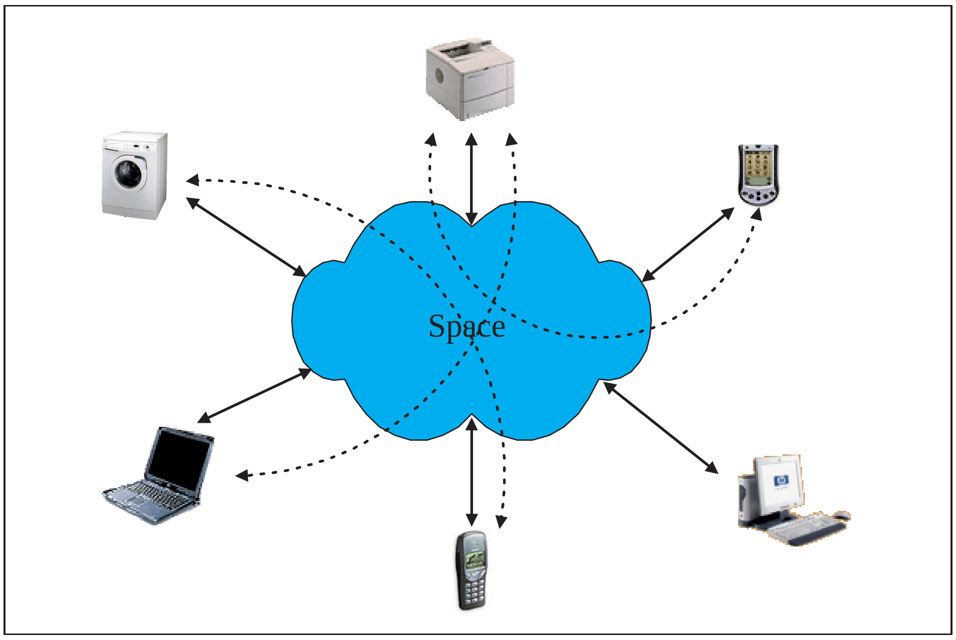

Figure 1: χ-Spaces Architecture

2 The χ-Spaces framework

The χ-Spaces [CCM02b] framework is tailored to the development of security protocols. Its programming language is a direct implementation of the SPL (Security Protocol Language) [CW01], a formal model for the study of security protocols. The χ-Spaces framework is implemented in Java [GJSB00] and as a consequence enjoys the portability typical of Java applications. The architecture of χ-Spaces rests on the general notion of network middleware or tuple space which has been first introduced in the mid-1980's by the project Linda [Gel85]. In this setting a space should supports three principles: anonymous communication, universal addressing and persistence of information.

2.1 Architecture

The χ-Spaces architecture is sketched in Figure 1. Communication between principals occurs via a *space*, a *remotely available* storage area. The approach of χ-Spaces does not rely on any specific space implementation; any modern coordination framework can be plugged in by writing an appropriate driver. The standard distribution of χ-Spaces provides a driver for IBM TSpaces [LCX$^+$01]. Stepping away from point-to-point communication, χ-Spaces is ideal in a highly dynamic environment, such as, for instance, Jini federations [CCM02a, Sun00]. Cryptographic primitives are not embedded either and so χ-Spaces can be used with any Java Cryptographic Architecture (JCA) [Sun] compliant providers. End-users need not to trust any particular implementation.

A χ-Spaces program can be either *interpreted* or *compiled* to JAVA code, which can then be refined via inheritance. An useful addition is the possibility to generate a JAAS [LGK+99] module implementing the protocol. Further details on the χ-Spaces programming methodology can be found in [CM02] and [CCM02b].

2.2 Programming language

Security protocols are frequently described as a sequence of message between distributed principals. A very simple but yet effective example is the *ISO 1-Pass Symmetric Key Unilateral Authentication* protocol [CJ97]:

$$A \to B : \quad \{Na, B\}_{Key(A,B)}$$

In this protocol the initiator A sends a message to the responder B. The curly brackets indicate encryption – in this example under the symmetric key shared by A and B. The value Na, the nonce, is meant to be fresh and unguessable. Its purpose is to ensure that the request of A is recent. For more sophisticated examples, descriptions of this kind are not enough to describe all the details of a protocol. A principal for example may decide a course of execution depending on the information contained in the messages received.

In this section we briefly describe the syntax for χ-Spaces programs – for more details refer to [CCM02b]. The main instructions are:

- `new` generates a nonce,
- `in` takes a message from the space, and
- `out` sends a message to the space.

A sequence of instructions separated by dots forms a sequential process. A χ-Spaces program consists of a parallel composition of sequential processes. There is no assumption on the order of execution of parallel components. The language supports an operator for unbounded replication as well, the process term `!P` executes as an unbound parallel composition of the process `P`.

The χ-Spaces implementation of the simple protocol above is given in Figure 2. It illustrates the use of parametric definition of sequential processes – in the example `Init` and `Resp`. The keys used along a sequential process must be declared in the preamble of its definition. In the example only the principals `"Alice"` and `"Bob"` are involved. In order to extend this protocol to allow another initiator, say `"Eve"`, is enough to replace the last line in Figure 2 with

```
def
 Init(A, B) := {Key(A, B)}
          new(Na) .
          out {(Na,B)}Key(A, B);

 Resp(A, B) := {Key(A, B)}
         in [*C Key(A, B) > (X, B)];
end

 Init{"Alice"}{"Bob"} | Resp{"Alice"}{"Bob"}
```

Figure 2: *ISO 1-Pass Symmetric Key Unilateral Authentication* in χ-Spaces

`Init {"Alice","Eve"}{"Bob"}| Resp{"Alice","Eve"}{"Bob"}`.

In χ-Spaces different threads of execution are spawned for every possible combination of the actual arguments. In the program above the process `Init`, for example, is launched twice, one time with parameters `"Alice` and `"Bob"` and the other time with `"Eve"` and `"Bob"`.

2.3 Messages and patterns

A message in an output can be a single variable, a constant, a tuple of messages or an encryption. Examples of messages are `5`, `"Alice"`, and `(x,(2,3,"Bob"))`. An encryption consists of a message and a key, e.g. `{(1,20)}Pub("Alice")`. All variables occurring in a message must be bound, if not the parser issues an open term error message.

A pattern in an input acts as a binder over the free variables. An input successfully takes a message from the space if it matches against the pattern. Simple examples of patterns are `5`, `x` and `(x,(x,10))`. Non-linear patterns and nested tuples are permitted, in these cases the pattern matching is performed left-to-right in a depth-first fashion. A decryption consists of a variable for the cipher, a key expression and a pattern, for example `[*C Priv("Bob") > (1,w)]`. An input with a decryption pattern first reads an arbitrary cipher from the space, decrypts it by using the key and tries to match the result against the pattern; if successful, it removes the cipher from the space.

To properly deal with keys χ-Spaces introduces a simple type distinction: key variables (*e.g.* `$k`), basic value variables for constants like nonces or agents names and general variables (*e.g.* `*C`) for any value, inclusive constants, tuples, keys and ciphers. The pattern matching takes into account these types so for example `$k` only matches a key in the space.

3 Concerns in the design of security protocols for low-power wireless devices

The design of security protocols for low-power wireless devices, such as mobile phones, personal digital assistants (PDAs), etc. faces challenges which are not normally found when tackling more powerful devices.

Performance

An important aspect that protocol developers must keep in mind is performance. Although this is always the case, when dealing with small devices this concern grows significantly in proportions. Currently most devices cannot meet the performance requirements of *public key* cryptography, e.g. RSA. In [DB99] the authors show that signing a message using a 512-bit RSA key takes more that 7 seconds on a 16 MHz Palm Pilot. The performance issues become more severe in JAVA enabled mobile phones. Preliminary benchmarks we conducted on the Motorola Accompli 009 showed that encryption with a 512-bit RSA key took well over two hours to complete. This is not acceptable. The end results is that current technology rules out public key cryptography in the design of security protocols for low-power devices.

Clocks

A number of security protocols rely on the use of *time stamps*. Time stamps were introduced by Denning and Sacco [DS81] to avoid replay attacks. As a well-known example of a security protocol relying on time-stamps consider the Kerberos v5 authentication protocol [SNS88].

Although time stamps are used in popular security protocols for everyday tasks, they have been criticised [BM91, Gon92]. In general time stamps require the clocks of the principals involved in the protocol to be *synchronized*. Clock synchronization is a well-known problem of distributed computing [Tel94]. A number of protocols exist to address this issue. However the most popular of these protocols pose a security risk in the context of security protocols. Better protocols do exist [Mil88], however they require an underlying authentication to be present. It looks like the problem of chicken and egg [BM91].

Clock synchronization is not a severe limitation in a limited and relatively static environment of a *local area network* (LAN). Generally, communication in a LAN environment is reliable. In a wireless environment communication is much less reliable, this, coupled with the ever-changing network topology, makes the task of clock synchronization rather troublesome. From a pragmatic point of view clock synchronization is best to be avoided in a wireless environment.

3.1 Choosing an authentication protocol

The previous discussion gives us some discerning criteria when choosing an authentication protocol suited for low-power wireless devices, mainly:

Efficiency No public-key cryptography should be used on low-power devices.

Clocks No time-stamps should be used to avoid the burden of clock synchronization in a wireless environment.

Scalability The protocol should scale well to an environment with thousands of principals.

Maintainability The protocol will be deployed on a variety of devices. To ease the deployment task only standard widely available cryptographic primitives should be used: encryption, decryption and signing.

Recent research in security protocol design has addressed these issues. Protocols specifically designed for low-power wireless devices have been proposed, for example, in [WC01a, WC01b, JP01]. The protocol in [JP01] has, however, been found flawed in [WC01a]. An interesting protocol meeting our criteria is described in [WC01b]. The same authors propose a more scalable protocol in [WC01a], this latter, however, does not meet our requirements as it involves the use of public-key encryption on a low-power device. Furthermore the use of non-standard cryptographic operations would result in low-level coding and very little choices for the cryptographic provider.

4 Server specific MAKEP

In this section we describe a protocol for *mutual authentication and key exchange* (MAKEP) presented in [WC01b]. The protocol is called *server specific* MAKEP (SSMAKEP). Its distinguished characteristic is that it requires an asymmetric amount of computational power. More specifically, we distinguish between a *server* whose computational power is of workstation level and a *client* which is assumed to have a limited amount of computation power, of the order of a PDA or cellular phone. On the client side the protocol relies solely on symmetric key cryptography, on the server side the more expensive public-key cryptography is used instead.

In the following description of the SSMAKEP protocol we use the notation $A \rightarrow B : M$ to say that the message M is sent from A to B. Encrypted messages are denoted by $\{M\}_{Key(A)}$ and $\{M\}_{Pub(A)}$, as a special case the notation $\{M\}_{Priv(A)}$ stands for the message M signed by A:

$$\begin{array}{lll}
(1) & A \rightarrow Ta: & A, S \\
(2) & Ta \rightarrow A: & \{A, \{Key(A)\}_{Pub(S)}\}_{Priv(Ta)} \\
(3) & A \rightarrow S: & \{Ra\}_{Key(A)}, \{A, \{Key(A)\}_{Pub(S)}\}_{Priv(Ta)} \\
(4) & S \rightarrow A: & \{Ra, Rs, S\}_{Key(A)} \\
(5) & A \rightarrow S: & \{Rs\}_{Key(A)}
\end{array}$$

Initially (Step 1) the client A contacts a *trusted authority* Ta saying that it intends to communicate with S. As a response (Step 2), Ta sends to A a *signed certificate* which contains A's long term encryption key, encrypted by S's public key. This certificate contains all the information S needs to engage in the protocol with A. A can then contact S (Step 3) and send the certificate and a challenge: an encrypted nonce. Being the certificate signed, S can trust its content and extract the key it needs to communicate with A. S proceeds by answering A's challenge with another challenge in the form of the encrypted nonce Rs. A's answer to the challenge completes the protocol. In [WC01b] the authors claim that the SSMAKEP protocol can be proved secure in a security model similar to the one presented in [BR93].

5 An implementation in χ-Spaces

In this section we show how the SSMAKEP protocol can be implemented in χ-Spaces.

In a χ-Spaces program we need a process definition for each principal role in the protocol. Subsequently, actual principals are obtained as *instances* of the corresponding roles. The roles for the SSMAKEP protocol can be found in Figure 3. To get hold of the needed keys the principals will rely on a specified keystore, if a certain key cannot be found an error will be generated. From the client point of view, its keystore must contain only its secret key. The servers must know their own key and the trusted authority public key. The trusted authority has more responsibilities. Indeed its keystore must contain the long-term key of every client and the public keys of all the servers. It is fundamental that only *trusted* servers make it this far. Let **RS** be the set of trusted servers. Conceivably the trusted authority will do the necessary checks to ascertain the identity of every server before including it in **RS**. If a corrupted server is in **RS**, the protocol is subject to a man-in-the-middle attack (see Appendix A).

A successful run of the SSMAKEP protocol will be shaped as in Figure 4. This event diagram was generated using χ-Sim, a security protocol simulator which is part of the χ-Spaces framework.

```
def
 Client(A, B):= {Key(A)}
     out ("S", A, B).
     in ((A, *C1), *C2).
     new (Ra).
     out (B, (A, *C1, *C2), {Ra}Key(A)).
     in (A, [*C3 Key(A) > (Ra, Rb, B)]).
     out (B, {Rb}Key(A));
end

def
 Server(B) := {Pub(B), Priv(B), Pub("Ta")}
   ! in (B, (A, [*C1 Priv(B) > $k], [*C2 Pub("Ta") > (A,*C1)]),
         [*C3 $k > Ra]).
     new (Rb).
     out (A, {(Ra, Rb, B)}$k).
     in (B, [*C4 $k > Rb]);
end

def
 Ta() := {Key(A), Pub(B), Pub("Ta"), Priv("Ta")}
   ! in ("Ta", A, B).
     out  ((A, {Key(A)}Pub(B)),
          {(A, {Key(A)}Pub(B))}Priv("Ta"));
end
```

Figure 3: SSMAKEP roles in χ-Spaces

5.1 Performance evaluation

We provide some benchmarks for the χ-Spaces implementation of the SSMAKEP protocol. In this work we focus one of the top-performers in the PDA scene: the Compaq iPAQ.

The Compaq iPAQ supports the PersonalJava virtual machine. The current PersonalJava JVM (1.1.8) is a limited version of the standard 1.1 JVM. To put our data into perspective we used a low-end PC as reference. The machines used for our tests had the following configuration:

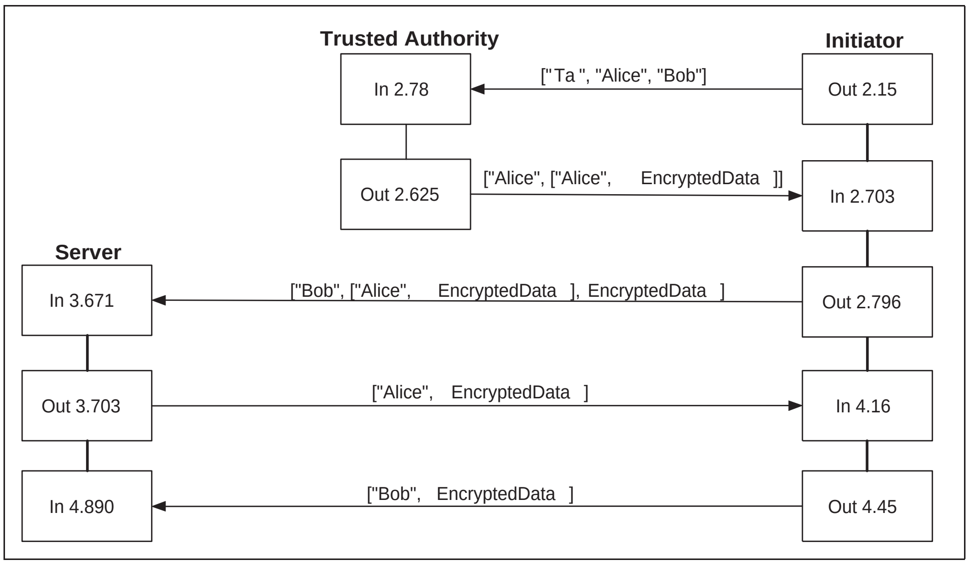

Figure 4: A successful MAKEP run

Model	Processor	Memory	Network	OS	VM
Compaq iPAQ H3660	206MHz Intel StrongARM	64MB RAM	11mbps IEEE 802.11b Wireless LAN	Familiar GNU/Linux	Sun PersonalJava 1.1.8
Low-end PC	Intel Celeron 300MHz	128MB RAM	10mbps Ethernet	Windows 2000	JVM 1.4

In Figure 5 we can see the performances of the protocol on the test machines. The cryptographic provider used was BouncyCastle 1.15[1], a popular open-source choice. The protocol was run using a 256bits TwoFish Cipher. Although the time spent doing cryptography is kept to unnoticeable levels, the protocol is still slow compared to a low-end PC, this is due to the slower processor and memory of the PDA. Indeed, marshaling and unmarshaling of the transmitted data account for most of the computation time. In Figure 6 we show the performances of the cryptographic part of the protocol for various ciphers, among which four of the five AES finalists[2]. Note that in this imple-

[1]Freely available from `www.bouncycastle.org`

[2]The low performances of Rijndael are due to an unoptimized implementation, the next release of the BouncyCastle provider will fix this.

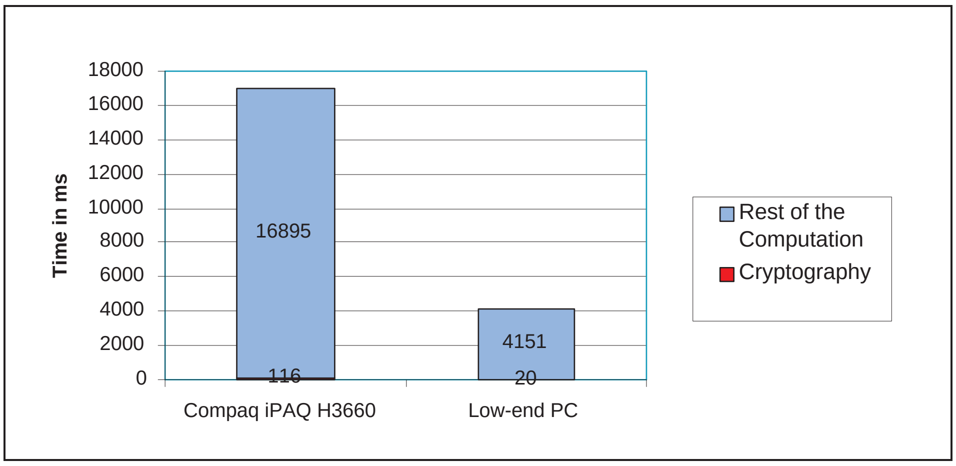

Figure 5: SSMAKEP Performances

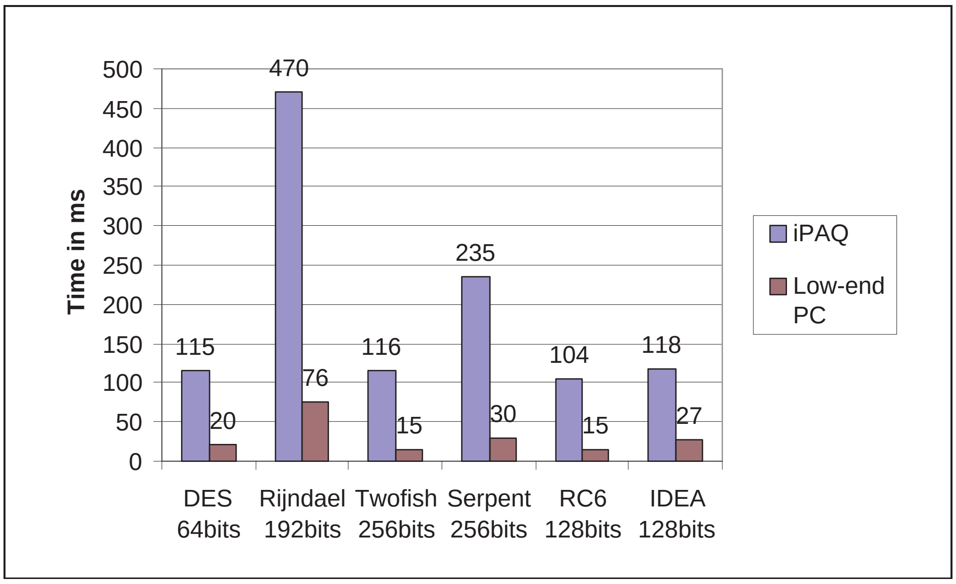

Figure 6: Cipher performances

mentation the symmetric key belonging to the client was stored on the PDA with no protection. Traditionally *password based encryption* (PBE) is used to implement a *secure keystore*. χ-Spaces, via the chosen JCE provider, fully supports keystores. However, even weak PBE is quite costly on a low-power device. The default JAVA keystore format (JKS), is secure against tampering (but not inspection). It relies on PBE based on SHA digests and TripleDES. In Figure 7 we show a detailed account of the performances of the SSMAKEP protocol on the Compaq iPAQ when the client's key is stored in a JKS keystore. At the moment secure keystores are way too costly at the PDA level. We believe that the security of the key stored in the PDA should be tackled at a different level, possibly securing the PDA itself.

It is possible to argue that JAVA is too slow to program cryptographic protocols on low-power devices. Indeed a C implementation of the same protocol would be significantly faster. However, in a real-life scenario, a service provider which needs to implement an authentication protocol, must support a variety of devices. Most likely, it must support *any* device. JAVA becomes soon an interesting option due to its portability which surpasses the down side of its performances.

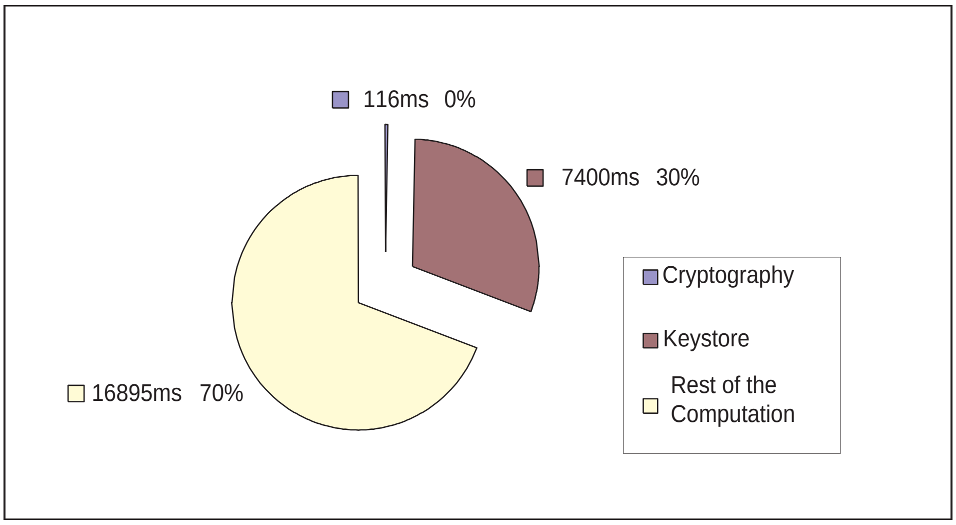

Figure 7: iPAQ Performance Details

6 Security properties

In this section we investigate some interesting security properties of the χ-Spaces implementation of the SSMAKEP protocol. We stress that the following discussion does not concern the *specification* of the protocol but its actual *implementation*. We are formal and precise in our study and make use of the net-semantics of SPL (see [CW01]) which can be adapted to χ-Spaces to state and prove security properties – a χ-Spaces program is easily translated into a corresponding SPL process (see [CCM02b]). The proofs of the properties we claim to hold for the χ-Spaces implementation of the SSMAKEP protocol follow from the dependencies among the various events that occur in a protocol run.

A run of the protocol is described by a sequence of configurations and events. Configurations $\langle p_i, s_i, t_i \rangle$ are triples consisting of a process term p_i which denotes the point of control reached by the protocol and determines the possible events that can occur from that point, a set s_i containing all the "new" (random) values that appeared so far in the run, and a set of messages describing the contents of the tuple space. When an event occurs a configuration evolves into another one. Events carry actions such as $act(e) = i : outnew\ n\ M$, the action of an output event e, where i is an index denoting the parallel component the event belongs to, n is a freshly created value, and M a message sent onto the tuple space. The action $act(e) = i : in\ M$ instead is that of an input event. Write $M \sqsubseteq M'$ when the message M is subterm of the message M' and if t is a set of messages write $M \sqsubseteq t$ if there is a message $M' \in t$ such that $M \sqsubseteq M'$. Let *SSMAKEP* be the *SPL* term for the SSMAKEP protocol system where **C** the set of names of agents that can be clients in the protocol. Let **RC** be the clients and **RS** the servers that are registered with the trusted authority and let **S** be the servers with which clients might want to communicate. We do not study the system in isolation but together with a spy which is a parallel component of *SSMAKEP*. The spy can eavesdrop on all the communication that passes through the space, encrypt and decrypt messages with the keys it gets hold of, and compose and decompose structured messages (see [CCM02b] for the χ-Spaces code of a possible spy). Let the SPL-process terms *Client*(A, B), *Server*(B), *Ta*(A, B) and *Spy* correspond to the respective χ-Spaces terms (this is done in a straightforward way as shown in [CCM02b]) and consider the following system:

$$\begin{array}{lcl}
P_{cl} & = & \|_{(A,B)\in \mathbf{C}\times\mathbf{S}}\ \mathit{Client}(A,B) \\
P_{sv} & = & \|_{B\in \mathbf{RC}}\ \mathit{Server}(B) \\
P_{ta} & = & \|_{A\in \mathbf{RC}}\, \|_{B\in \mathbf{RS}}\ \mathit{Ta}(A,B) \\
 & & \\
P_{spy} & = & \mathit{Spy} \\
 & & \\
\mathit{SSMAKEP} & = & \|_{i\in\{cl,sv,ta,spy\}} P_i
\end{array}$$

The security theorems in this section hold for every run of the *SSMAKEP* system:

$$\langle \mathit{SSMAKEP}, s_0, t_0 \rangle \xrightarrow{e_1} \ldots \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} \ldots$$

A central property of the protocol is the secrecy of a client's long term key. We show that the long term key of a client never appears in clear and by itself on the network, provided it is not leaked from the start and provided that all private keys of the registered servers are not corrupted.

Theorem 6.1 (Secrecy of client's long term key). *Let $A \in \mathbf{C}$ be a client such that $Key(A) \not\sqsubseteq t_0$. If for all registered servers $rS \in \mathbf{RS}$ the private key $Priv(rS) \not\sqsubseteq t_0$ then at every stage j of an SSMAKEP run $Key(A) \notin t_j$.*

In other words, under the conditions of the previous theorem, only registered servers get to know long term keys of registered clients. For example suppose that *Eve* $\notin \mathbf{RS}$ is not a registered server. Since *Eve* is not registered its private key could be corrupted and $Priv(Eve) \in t_0$. If *Eve* gets to know $Key(A)$ by receiving the message $\{Key(A)\}_{Pub(Eve)}$ then this can be eavesdropped by a spy. The spy can use the leaked private key of *Eve* to decrypt and publish $Key(A)$ on the network. This scenario is not admitted by the previous theorem. This concept can be made more precise using the notion of "surroundings" (see [CCM02b]). Then one can show that $Key(A)$ always appears in a message (is in the "surrounding" of) encrypted with the public key of a registered server.

The protocol aims at setting up a session key between registered clients and servers. The session key is built out of two parts, a server part and a client part. The next theorem shows when the protocol keeps the two parts of the session keys secret.

Theorem 6.2 (Secrecy of session key). *Let $A \in \mathbf{C}$ be a client such that $Key(A) \not\sqsubseteq t_0$. Suppose that for all registered servers $rS \in \mathbf{RS}$ the private key $Priv(rS) \not\sqsubseteq t_0$.*

Server part of session key *If the run contains a server event b_2 with action*

$$act(b_2) = sv : B : r : out\, new\, rb\ \ A, \{ra, rb, B\}_k$$

then at every stage j of an SSMAKEP run $rb \notin t_j$.

Client part of session key *If the run contains a client event a_3 with action*

$$act(a_3) = cl : A, B : l : out\, new\, ra\ \ B, A, M, M', \{ra\}_{Key(A)}$$

then at every stage j of an SSMAKEP run $ra \notin t_j$.

A property that is certainly desired for the *SSMAKEP* protocol is authentication on the server side. The server wants to be sure that it is indeed setting up a session key with the client and not with some malicious party that masquerades as the client, perhaps replaying an old message. Under the assumption that no registered server has a corrupted key one can ensure the following agreement property which is a strong form of authentication.

Theorem 6.3 (Server authenticates client). *Suppose that $Priv(rS) \not\sqsubseteq t_0$ for all $rS \in \mathbf{RS}$ and that $Priv(Ta) \not\sqsubseteq t_0$. If a run of SSMAKEP contains server events b_2, b_3 of a registered server $B \in \mathbf{RS}$ with actions*

$$\begin{array}{rcl} act(b_2) & = & sv : B : r : out\,new\,rb\ A, \{ra, rb, B\}_k \\ act(b_3) & = & sv : B : r : in\ B, \{rb\}_k \end{array}$$

where $A \in \mathbf{C}$ such that $Key(A) \not\sqsubseteq t_0$ then $A \in \mathbf{RC}$ and $k = Key(A)$ and the run contains client events a_3, a_4, a_5 with actions

$$\begin{array}{rcl} act(a_3) & = & cl : A, B : l : out\,new\,ra\ B, A, M, M', \{ra\}_{Key(A)} \\ act(a_4) & = & cl : A, B : l :\in\ A, \{ra, rb, B\}_{Key(A)} \\ act(a_5) & = & cl : A, B : l : out\ B, \{rb\}_{Key(A)} \end{array}$$

for some messages M, M'.

If registered servers behave according to their code and do not have corrupted private keys, then a strong from of authentication also holds for the client with respect to the server.

Theorem 6.4 (Client authenticates server). *Suppose that $Priv(rS) \not\sqsubseteq t_0$ for all $rS \in \mathbf{RS}$. If a run of SSMAKEP contains client event a_4 with action*

$$act(a_4) = cl : A, B : l : in\ A, \{ra, rb, B\}_{Key(A)}$$

where $A \in \mathbf{C}$ such that $Key(A) \not\sqsubseteq t_0$ then $A \in \mathbf{RC}$ and $B \in \mathbf{RS}$ and the run contains a server event b_2 with action

$$act(b_2) = sv : B : r : out\,new\,rb\ A, \{ra, rb, B\}_{Key(A)}$$

One of the weak points of the SSMAKEP protocol is that all registered servers need to be uncorrupted so that their private key is not leaked to a malicious agent. It is enough that only one server gets corrupted to make all the above security properties fall. In particular if one server leaks the key to a spy, then the spy can get hold of the long term key of any registered client an therefore can get hold of the session keys and act as a registered user instead of anyone else.

7 SSMAKEP protocol use case & deployment

A company T is advertising the services of a number of registered providers **RS**. To clients that register with T access to the services of providers in **RS** is granted under particularly convenient conditions. For example, T is a travel agency promoting a number of hotels. The clients that register with T (e.g. paying a one time membership fee) will be able to browse, with their last generation mobile phones, a list of available hotels and reserve a room at a special rate. Both clients and hotels can decide to join and leave T. The travel agency T will register new hotels following a certain policy and, for example, guarantee a certain quality of services within **RS**. We seek an authentication protocol that can cope with this dynamic environment and ensures that:

1. Unregistered clients won't obtain the special discount when trying to reserve a room from a hotel in **RS**.

2. Clients that book a room from a hotel that appears in **RS** want to be sure the hotel is indeed registered with T to avoid unpleasant surprises on their final bill.

The SSMAKEP protocol, as we saw, has the necessary authentication guarantees and its client side is light enough to sit on a mobile phone. The danger of a corrupted hotel manager that impersonates a client to get a discount for his own holiday is acceptable if, for example, T promises to take actions against a discovered misbehaving hotel (in addition, client membership keys could expire after some time and reissued whenever a corruption is detected). SSMAKEP provides clients with a *check-in token Rs* that can be used at check-in time to prove that an advanced booking took place.

Deployment of a χ-Spaces SSMAKEP implementation

The client. Alice wants to become a member of T and goes to T's office to purchase the membership. The travel agency registers Alice in **RC**. Alice gets from T the χ-Spaces SSMAKEP client software which she can easily install on her Java-enabled mobile phone. She will also get a membership key ($Key(A)$) to be stored on her phone. Alice is now a registered member of T and her phone enabled for making convenient reservations with the hotels advertised by T.

The server. Hotel B wants to be promoted by T and asks for registration. If T accepts, it includes B in **RS** and provides B with the χ-Spaces SSMAKEP server software, which includes a tuple space. This software is JAVA based and can easily be installed on B's system. Moreover T and B exchange public keys. The hotel wants to place the SSMAKEP server on a machine which is protected by a firewall. To enhance security the space could be run *outside* the firewall – the server software needs only one fixed IP address to communicate with the space, however, clients can conveniently connect from various, possibly dynamic, IP addresses to the space.

8 Conclusions

Starting with the goal of deploying security protocols for low-power wireless devices we moved on to find an authentication protocol suitable for this environment. Preliminary considerations led us to some discerning criteria in our search, chiefly we wished to avoid public-key cryptography and time-stamps. A suitable candidate for our purposes is the SSMAKEP protocol. We provided an implementation in χ-Spaces and analysed its performances on the popular Compaq iPAQ PDA. Our tests showed performances which, relatively speaking, could be acceptable. We noted, however, that the traditional keystore implementations are, at the moment, too expensive to be used.

Using the underlying SPL model of χ-Spaces we studied the correctness of our SSMAKEP implementation. At the moment we are busy collecting benchmarks for other low-power devices, such as the Palm m500, the Sony Clie and the Sony-Ericsson P800. We are also studying the performance gap between native code and Java. An interesting area we are exploring is the use of Jini to discover the space to be used as communication medium. The recently available PersonalJava 1.2 platform provides all the necessary features to interface with Jini, this comes, however, at the price of higher memory requirements.

Acknowledgements

We would like to thank COGS, University of Sussex, for granting access to the Compaq IPAQs used to perform the benchmarks reported in Section 5.1. Our thanks to Matt Newsome who set up the IPAQs for remote access and provided us with all the information we needed to get the job done.

References

[ACR00] ACROS. Bypassing warnings for invalid SSL certificates in Netscape Navigator. ACROS Security Problem Report, 2000.

[BM91] Steven M. Bellovin and Michael Merritt. Limitations of the Kerberos authentication system. In *USENIX Conference Proceedings*, pages 253–267, Dallas, TX, USA, 1991.

[BR93] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Proceedings CRYPTO '93*, volume 773 of *LNCS*, pages 232–349. Springer-Verlag, 1993.

[CCM02a] Mario Cáccamo, Federico Crazzolara, and Giuseppe Milicia. χ-Spaces: Authentication for Jini services, June 2002. Presentation at the 6th Jini Community Meeting. Available from www.chispaces.com.

[CCM02b] Mario Cáccamo, Federico Crazzolara, and Giuseppe Milicia. χ-Spaces: From a model to a working language. Technology White Paper. Available from www.chispaces.com.

[CCM02c] Mario Cáccamo, Federico Crazzolara, and Giuseppe Milicia. The ISO 5-pass authentication in χ-Spaces. In *Proceedings of the Security and Management Conference (SAM)*, pages 490–495, Las Vegas, June 2002.

[CJ97] John Clark and Jeremy Jacob. A Survey of Authentication Protocol Literature: Version 1.0. www-users.cs.york.ac.uk/~jac, 1997.

[CM02] Federico Crazzolara and Giuseppe Milicia. Developing security protocols in χ-Spaces. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems (NordSec)*, November 2002. To Appear.

[CW01] F. Crazzolara and G. Winskel. Event in security protocols. In *Proceedings of the Eight ACM Conference on Computer and Communications Security*, Philadelphia, 2001.

[DB99] N. Daswani and D. Boneh. Experimenting with electronic commerce on the PalmPilot. In *Proceedings of Financial Cryptography 1999*, volume 1648 of *LNCS*, pages 1–16. Springer-Verlag, 1999.

[DS81] D. E. Denning and G. M. Sacco. Time-stamps in key distribution systems. *Communications of the ACM*, 24(8):533–536, August 1981.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* Sun microsystems, June 2000.

[Gon92] Li Gong. A security risk of depending on synchronized clocks. *Operating Systems Review*, 26(1):49–53, 1992.

[JP01] Markus Jakobsson and David Pointcheval. Mutual authentication for low-power mobile devices. In *Proceedings of Financial Cryptography 2001*, volume 2339 of *LNCS*. Springer-Verlag, 2001.

[LCX^{+}01] Tobin J. Lehman, Alex Cozzi, Yuhong Xiong, Jonathan Gottschalk, Venu Vasudevan, Sean Landis, Pace Davis, Bruce Khavar, and Paul Bowman. Hitting the distributed computing sweet spot with tspaces. *Computer Networks*, 35(4):457–472, 2001.

[LGK^{+}99] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. User authentication and authorization in the Java platform. In *15th Annual Computer Security Applications Conference*, pages 285–290. IEEE Computer Society Press, 1999.

[Low96] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *TACAS'96*, volume 1055 of *LNCS*, pages 147–166, 1996.

[Mil88] D.L. Mills. Network time protocol. RFC 1059, 1988.

[NS78] Roger M. Needham and Michael D. Schröder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), December 1978.

[RTVH01] Roger Riggs, Antero Taivalsaari, Mark Vandenbrink, and Jim Holliday. *Programming Wireless Devices with the Java(TM) 2 Platform (Micro Edition)*. Sun microsystems, 2001.

[SNS88] J. Steiner, C. Neuman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of Winter USENIX Conference*, Dallas, TX, USA, 1988.

[Sun] Sun microsystems. *JavaTM Cryptography Architecture API Specification and Reference*. java.sun.com.

[Sun00] Sun microsystems. *Jini Architecture Specification*, October 2000. Version 1.1. Available at www.sun.com/jini.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[WC01a] Duncan S. Wong and Agnes H. Chan. Efficient and mutually authenticated key exchange for low power computing devices. In *Proceedings of ASIACRYPT 2001*, volume 2248 of *LNCS*. Springer-Verlag, 2001.

[WC01b] Duncan S. Wong and Agnes H. Chan. Mutual authentication and key exchange for low power wireless devices. In *Proceedings of IEEE Milcom*, McLean, VA, (USA), October 2001.

[WL94] Thomas Y. C. Woo and Simon S. Lam. A Lesson on Authentication Protocol Design. *Operating Systems Review*, pages 24–37, 1994.

A Compromised Server Scenario

As briefly mentioned in Section 5 and Section 6, if a server in **RS** is compromised the SSMAKEP protocol is not secure anymore. In Figure 8 we can see a trace of the resulting attack. An attacker (Mallorie) which knows the private key of a compromised server is able to spoof the secret key of any client attempting to authenticate with the server. Subsequently Mallorie is able to impersonate the client.

A different issue is that servers have always the possibility of impersonating the clients they serve. This is normally not an issue, however, in certain situations it might be problematic, e.g. the clients need to fully trust the servers with their identities.

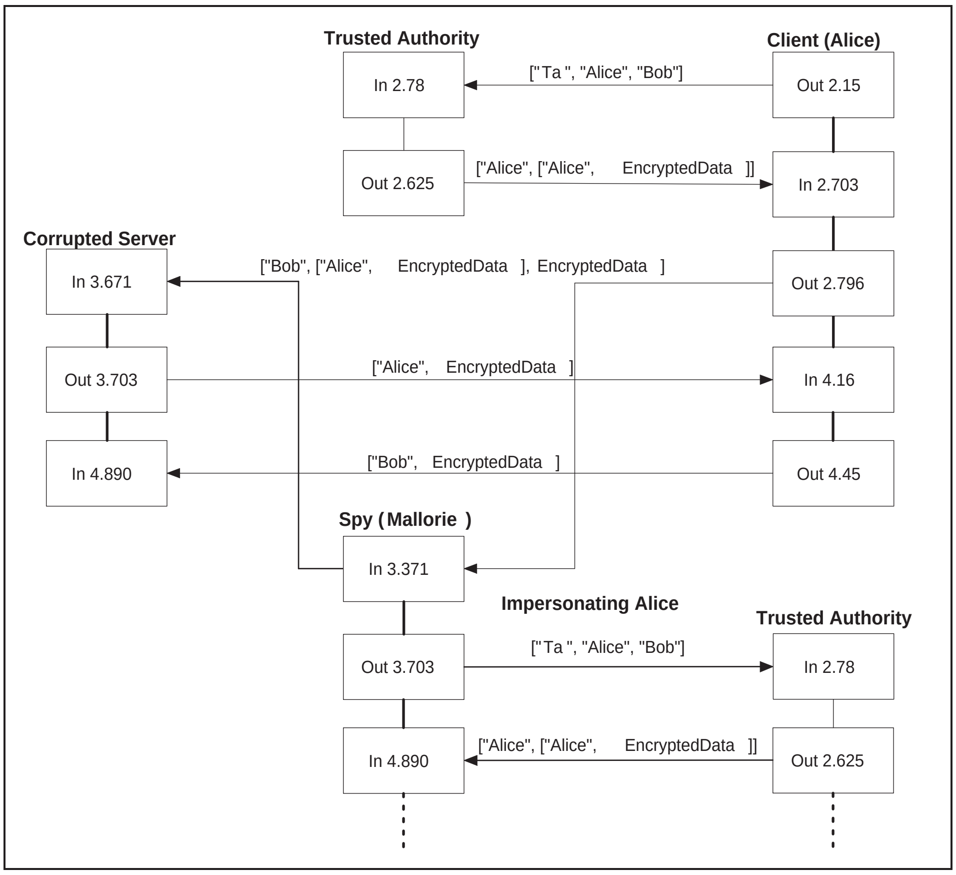

Figure 8: Compromised Server Attack