IMPLEMENTING A SECURE LOG FILE DOWNLOAD MANAGER FOR THE JAVA CARD

Constantinos Markantonakis*, Simeon Xenitellis†
Information Security Group, Royal Holloway,
University of London, Egham, Surrey, TW20 0EX, UK
{C.Markantonakis, S.Xenitellis}@rhbnc.ac.uk

Abstract Current smart card technology suffers from inadequate log file handling mechanisms. While there are a number of theoretical security protocols integrating smart cards in a variety of systems, there are very few test implementations. In this paper we present details of an experimental implementation of a secure log file download protocol on two different Java Card platforms. We also provide a discussion of the performance results along with our experiences from implementing the theoretical design in a real multi-application smart card environment.

Keywords: Security, Smart card, Java Card, Multi application smart cards, Log files, Audit files.

1. INTRODUCTION

Among the mechanisms that help monitor system activity are audit log services. Such services, implemented either in kernel space, user space or, most commonly, a combination of both, can be used for purposes such as billing and debugging and, most importantly, in security related applications. An important security related application is the monitoring of system activity to identify security breaches and determine who is to be held accountable. In a smart card environment, and particularly in today's multi-application smart cards, log files are important in identifying breaches in system security by maintaining a database of the actions that the card has carried out. Smart card log files could be used to log

*The author's research is funded by Mondex International Limited. This work is the opinion of the author and does not necessarily represent the view of the funding sponsor.
†The author is funded by the State Scholarships Foundation of Greece.

B. Preneel (ed.), *Secure Information Networks*

operating system or application information [7] or even the results of card intrusion detection mechanisms.

With the introduction of true multi-application smart cards [3, 5, 12], the amount of information to be logged increases substantially. First, each transaction adds to the audit log file and second, multiple applications allow less file space for the log file(s). At the same time, since more than one application will reside on the card, there are many security related events to be logged.

An obvious question that needs to be answered is what happens when the log file space becomes full. The currently favoured approach suggests that the log files should be overwritten in a cyclic mode. In [6] we proposed that the log files should be securely downloaded to some other medium which will not suffer from immediate storage restrictions.

In this paper we give a secure log file download protocol, and describe results from a test implementation. The protocol uses minimal cryptography (as explained in the next section). It could also be considered as a reference point when designing more complex smart card applications.

For the protocol implementation we used two of the most popular Java Card API Ver 2.0 [8, 9, 10] compliant implementations currently available in the market, namely the GemXpresso Java Card [3] from Gemplus and the Cyberflex Open 16K Java Card [12] from Schlumberger. Notably, both these implementations allow dynamic application download and provide application isolation. Although Java Cards offer multi-application capabilities, there are still certain limitations such as the limited size of EEPROM and RAM, the limited processing power, and the lack of cryptographic functions in general. These factors forced certain design decisions which will become evident below.

This paper serves two purposes. First it provides performance measurements for the log file download protocol, and thus proves the concept is feasible. Secondly, it highlights issues which are relevant when developing real life Java Card applications. This last point is of particular importance since there are very few Java Card applications available today and we expect a substantial increase in the future.

The remainder of this paper is organised as follows. First, we outline the main characteristics of the log file download protocol. Subsequently we present the available smart card technology along with its main limitations. The next two sections describe the implementation details for the two platforms. Finally, we provide our concluding remarks and directions for further research.

2. AN OVERVIEW OF THE LOG FILE DOWNLOAD PROTOCOL

The following protocol is the simplest of the three presented in Cardis 98 [6]. The principal participants are as follows.

1. C (Card) represents the smart card. Typically, this is a tamper-resistant device, and has access to a variety of cryptographic algorithms and a good pseudorandom number generator.
2. ALSS (Audit Log Storage Server) receives and stores the transmitted log files. Depending on the environment the ALSS could be a "smart wallet", a personal computer used in conjunction with the smart card, or even a repository server connected to the Internet.
3. LFDM (Log File Download Manager) resides in the card as the only entity authorised to access the log files and implement the log file download protocol.
4. PD (Personal Device) i.e. the smart card reader/writer or personal wallet used by the card to communicate with the outside world.
5. A (Arbitrator) is responsible for receiving the downloaded log files in case of a dispute, and subsequently informing the entities involved of its decision.

In the implementation described here the ALSS is implemented as a PC-resident client application, and the LFDM is implemented as a smart card application. In practice the LFDM could also be a smart card operating system entity.

Prior to downloading any log files we require that the cardholders will be authenticated by their cards. An existing unilateral authentication scheme can be used although its details are not within the scope of this paper. The details of evidence generation are described in [1] and are also outside the scope of this paper.

The protocol involves the LFDM sending message 1 to the ALSS and the ALSS responding with message 2.

$$LFDM \longrightarrow ALSS : M_n \parallel f_{K_{AC}}(M_n) \tag{1}$$

where

$$M_n = (F \parallel T_c \parallel N_c \parallel I_{AC} \parallel CBind_n)$$

$$CBind_n = h(h(M_{n-1}) \parallel N_C)$$

$$ALSS \longrightarrow LFDM : SBind \tag{2}$$

where

$$SBind = (h(F) \parallel \text{“Log File Received”} \parallel T_S)$$

We assume that the card shares a key (K_{AC}) with an arbitrator. This key could be stored in the card during the personalisation phase (preferably) or securely updated at any later stage during the life cycle of the card. Additionally we assume that the card maintains a sequence number N_C which cannot be modified by any external events.

The notation used is as follows: $h(Z)$ is the one-way hash of data Z using an algorithm such as SHA-1, $f_K(Z)$ is the output of MAC function f using as input secret key K and data string Z. The log file data is represented by F, T_C is a newly generated time-stamp, N_C is the current sequence number incremented by one, and I_{AC} is the unique key identifier for key K_{AC} (telling the arbitrator which key to use to verify the MAC of the message). The $CBind_n$ variable links the previously sent message (M_{n-1}) with the current message sequence number (N_C). After sending 1 the LFDM stores in the card (in protected memory) a hash of the current file (F) and a hash of the current message sent (M_n). Note that the LFDM has not yet flushed the space occupied by the log file.

On receiving message 1 the ALSS informs the cardholder, e.g. by displaying a message on the PD's screen, that the message has been successfully received. Subsequently it extracts the log file (F) from message (M_n). The ALSS reply to the card consists of message 2.

On receiving the response message, the card checks for the correct hash value of the log file (F). If a correct hash value of the log file is present, the LFDM flushes the memory space used by the log file and replaces the previous value of M_{n-1} with the current value of M_n.

3. DESIGN

Software solutions that use smartcards can be easily separated into the client-side program and the smartcard applet. In our case, in order to implement the log file download protocol, two distinct entities have to be developed. The first one will reside in the card (that is, the LFDM) and the second will reside in the user's PC (the ALSS). In this section, we outline the technology available for implementing these two entities. Subsequently, we present the limitations and architectural characteristics, to give the reader an understanding of the issues involved.

3.1 THE CLIENT APPLICATION

The client application will implement the ALSS residing on the users PC. For the client side of the application, there are currently two interfaces available that enable the client to interact with the Java Card. These interfaces provide a level of abstraction to the developer. For example, with both interfaces the developer need not worry about how to access the particular smart card reader.

- PC/SC [15] was developed by Microsoft, Hewlett-Packard, Siemens-Nixdorf and the smart card manufacturers. PC/SC is tied to the Windows platform and typical supported programming languages are Visual Basic and various C++ compilers. This is the most widely used and supported architecture. Most smart card manufactures provide drivers for PC/SC.

- Another more recent initiative is the OCF (OpenCard Framework) [13], which enables Java applications to communicate with the smart card in a transparent and portable fashion. OCF is written in Java and was primarily developed by IBM and other computer technology providers. Nowadays, OCF enjoys support from Java Card manufacturers to banking organizations. It permits the client applications to access the smart card irrespective of the host operating system and CAD (Card Acceptance Device or Card Terminal). Support for OCF is currently becoming available on an increasing number of Java Cards and Card Terminals. Note that OCF and PC/SC do not require a Java Card; they both also work with non-Java smart cards.

3.2 SMART CARD APPLICATION DEVELOPMENT TOOLS

As previously stated the LFDM will be an application residing in the smart card. Two widely known development tools that allow preprocessing, downloading and execution of Java Card applets are as follows.

- The Gemplus GemXpresso Rapid Applet Development (RAD) Kit [3], is a Java-based application development environment that enables developers to write and test Java Card applets. The GemXpresso applet-prototyping card is the first smart card implemented on a 32-bit RISC processor. It is Java Card API 2.0 compatible but it also differentiates itself with a cut-down version of an RPC (Remote Procedure Call) [1] style protocol called DMI (Direct Method

Invocation) [14]. It is ISO 7816-2, -3, -4 compatible and it has 10KB and 5KB of heap memory available for applications.

- The Cyberflex Open 16K [2] is Java Card API 2.0 compliant and compatible with ISO 7816-2, -3, -4. It can accept applets with a total size of 16 KB including the heap, and it has a stack size of 128 bytes. It uses an 8-bit processor.

The developer creates applets using a typical Java development environment, pre-processes them using the development environment tools, and downloads them to the Java Card. Development environment tools enable communication with the running applet in the Java Card. Java Card simulators are included in both environments.

3.3 LIMITATIONS OF THE SMART CARDS AND THE DEVELOPMENT KITS

During the development of the test applications, we encountered the following limitations, that had to do either with the smart card capabilities or the Development Kits that accompanied them.

Of the two Java Cards, only the GemXpresso supports garbage collection. However, garbage collection, by nature, does not take place immediately after the memory ceases to be used, but after some implementation-specific delay. Also, the applications running on a smart card have a very short lifetime, so that cases of garbage collection that do not occur at all will probably be commonplace. Furthermore, in many JVM implementations, garbage collection takes place when memory gets exhausted and such a procedure would adversary affect the potential speed of the given Java Card. Due to these issues, special attention had to be given when coding not to use local variables and not dynamically allocate memory in frequently called functions. In extreme cases, "frequently" is defined as twice or more. When a local variable is instantiated, the memory of the stack is used. When a dynamic allocation is requested, the memory of the heap is used. In both cases, after the exit of the function, no memory is claimed back, and we gradually become short of memory. The solution is to use global variables, and reuse them as much as possible within the applet. It is desirable to restrict memory allocation to inside the constructor of the applet, because this is a guaranteed location that is executed only once, and it is a location where the memory has not become fragmented. We must note that garbage collection is not a prerequisite for Java Card 2.0 API conformance.

When sending data (Application Protocol Data Units [11]) from the card to the client and vice versa, a limit of approximately 256 bytes exists. In order to solve the problem, special programming has to be

used to send the data in blocks. This slows the applets significantly when communicating with the client, because switching from receiving to sending, and vice versa, is a slow procedure.

The (lack of) availability of cryptographic functions on the Java Cards is a two-fold problem. First, the export control restrictions imposed by many governments dissuade manufacturers from implementing them. Secondly, the JVM already takes up much of the resources of the Java Card, and manufacturers need to invest heavily to put the cryptographic core with the rest of the functionality. Both Java Cards used do not currently support real cryptographic functions.

The development environment usually offers a "simulator" that enables the programmer to easily test the applet without downloading it to the Java Card every time it is to be executed. In the case of the development kit of the Open16K, the simulator was not reliable enough, and attempts to use it had to be abandoned. The GemXpresso Java Card simulator was significantly better and quite usable.

The time from the generation-compilation of the applet on the development environment until the applet is running on the Java Card was computed to determine the compile-to-run cycle. It was noticed that the resulting times were between one and two minutes. These times are very long and in some cases involve a series of repetitive steps. This often leads to errors such as failing to correctly update the applet on the Java Card and carrying out tests on the previous applet. This was the case with the development environment of the Open 16K card.

3.4 DESIGN GOALS

The following are among the design principles followed while implementing the log file downloading protocol.

- Ensure that the log file download protocol will start and the legitimate smart card holder will approve it.
- Effectively identify restricted smart card resources in order to allow more adequate application design.
- Perform application code optimisation in order to achieve better application performance.
- Design dummy cryptographic primitives, since most of the cards do not offer such functionality (and when offered it is not fully accessible, as explained later).
- Evaluate the usability of the Java Card development tools.
- Examine application execution performance for both platforms.

3.5 COMMON IMPLEMENTATION DETAILS

Because the cryptographic functions were unavailable, dummy functions were implemented to cover the lack of hash and MAC functions. The input to the dummy MAC and hash functions was a buffer of arbitrary size and the output was a 16-byte string. The dummy hash was implemented using the modulus operation over the input several times and the dummy MAC was implemented using the dummy hash by adding an "exclusive or" with a key value.

Although log files of 1–2Kb may be more desirable [7] we used smaller files along with simple cryptographic functions for the following reasons.

It is rather difficult to manage large files (more than 512 bytes) on the card, and problems were encountered when reading a single file larger than 248 bytes from the card. It would also be time-consuming to perform cryptographic functions on large blocks of data. Additionally it would further delay the protocol execution since more than one log file packet would need to be transmitted (due to the 248-byte restriction on the APDU data buffer).

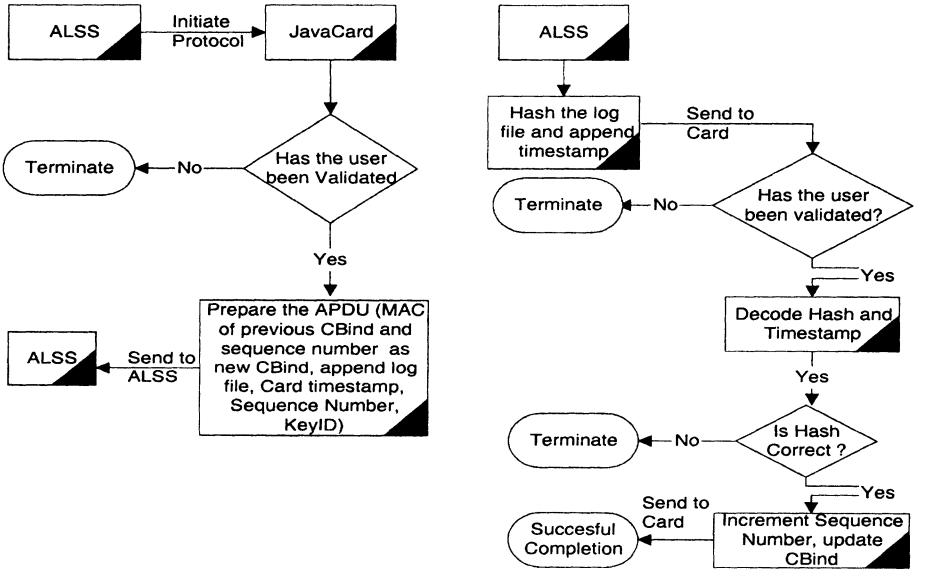

Figure 1 The two steps (SendData, VerifyReply) of the log file download protocol

Another issue that forced the common choice of log file size as 40 bytes, and such simple dummy cryptographic functions, was the poor performance of the card reader for the Cyberflex Java Card. As described

in §5.1, this card reader does not support long processing in the Java Card due to the desynchronisation of client/card communications.

The log file information could have been spread across more than one file to accommodate the limited abilities of the Java Cards. However, such a solution was not adopted, as it would require changes to the log file download protocol and generally to the security design of the system. Therefore, it was decided that the log file should be relatively small (248 bytes when running on the simulator and 40 bytes when running on the card). As previously mentioned, it was impossible to make the protocol work (on the card) when the log file was more than 40 bytes. However, we believe that with further code optimisations and more effective smart card memory usage the protocol would work with larger files. The steps of the log file download protocol are presented in figure 1.

Note that for the protocol to be implemented, the following functionality needs to be available. The card has to implement a hash and a MAC function (§4.3 and §5.3). The card must also be capable of storing certain transient information (N_C, I_{AC}) while waiting for the ALSS's reply. The ALSS has to implement the same hash function as the card.

The protocol functionality is encapsulated within three essential functions: first, the Validate operation that accepts the PIN and compares it with a default one, second, the SendLogFileData operation that implements the part of the protocol that sends, among other information, the log file data, and third the ReceiveALSSReply that implements the last step of the protocol.

4. GEMXPRESSO IMPLEMENTATION

In this section we present the implementation details when using the GemXpresso development kit.

4.1 GEMXPRESSO CHARACTERISTICS AND LIMITATIONS

Note that, although GemXpresso contains certain dummy cryptographic functions (DES, triple DES, MAC), these were not used. Instead, the dummy functions specified in §3.5 were used in both implementations to improve the comparison between the two experimental platforms.

Within the GemXpresso development kit a simulator is also provided. This simulator allows the programmer to experiment with the Java Card applications prior to downloading to the card.

4.2 IMPLEMENTATION

In this section, we present a more detailed design for the GemXpresso Java Card application. The application is written according to the DMI specification in Java. In order to maintain compatibility with the Open 16K platform did not use any 32-bit data types (specifically in the hash and MAC functions). We have shared the functionality of the log file download protocol between an applet and a library.

The Java Card applet contains all the functions accessible to the outside world (i.e. the client). These functions provide an external interface for the functions (defined in the library) directly accessing the log file.

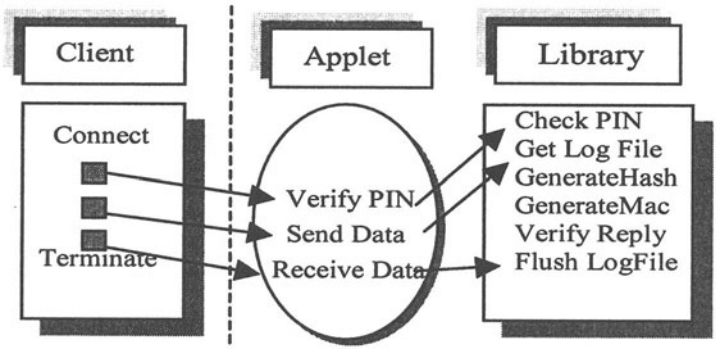

Figure 2 The architecture of the LFDM residing on the GemXpresso card and the Client application residing on the user's PC

In this architecture the sensitive log file information will only be accessed through the library classes, and a malicious client application cannot directly call the procedures accessing the log file. Because the library contains the basic log file access procedures (such as a log file API), these functions can then be shared among other trusted Java Card applications.

To see the advantages of this architecture, suppose the log file download protocol is to be upgraded — a new Java Card applet, implementing the new protocol and using the existing library, can be loaded into the smart card, without endangering the library functions. Also a potential log file library (Log File API) owner could easily "sell" access rights from the library to other smart card application programmers.

The other half of the log file download protocol (the client application) was written using Microsoft J++ and Microsoft Java SDK Ver 2.02.

4.3 RESULTS AND PERFORMANCE EVALUATION

Different results were generated depending on the actual size of the log files and whether the application ran on the card or on the simulator. The implementation and the testing of the client and LFDM application used a 400Mhz PC with 128 MB of RAM under Windows NT. When the client was communicating with the Java Card, the results were largely PC speed independent because the operations were mostly I/O intensive.

The results from ten consecutive protocol executions on the simulator with log file sizes of 248 and 40 bytes are presented in table 1. The "Connect" figure indicates the time spent by the client application connecting to the reader or the simulator. The next three values indicate the time spent in the following procedures: "VerifyPIN" verifies the user password, "SendData" forms the packets as defined by the protocol, and "VerifyReply" verifies the ALSS reply. The "Disconnect" figure represents the time spent closing the connection with the reader or the simulator.

Table 1 GemXpresso simulator results using log files of 248 and 40 bytes

	Simulator: 248 byte files		Simulator: 40 byte files	
Part name	*Time (ms)*	*Std.Deviation*	*Time (ms)*	*Std.Deviation*
Connect	11246	34.62	11246	24.18
Validation	210	4.97	210	4.20
Send Log File	200	0.42	140	0.32
Verify Reply	200	0.52	200	0.48
Disconnect	401	0.47	400.5	0.50
Total	12257	40.99	12197	29.68

It is worth mentioning that the "Connect" figure (for the simulator) is not very accurate. This is because, when the client is connecting to the simulator and tries to select the applet, in most cases it returns "false" (i.e. that the applet could not be selected). Surprisingly though, the applet execution continues as if the applet was properly selected.

From the figures in both tables we observe that the "Connect" and "Disconnect" values are almost identical. Similarly, the "VerifyPIN" procedure consumes the same amount of time in both cases. The "SendData" function in the second case takes less time since less data are involved in constructing the protocol packets.

The main reason for providing execution results on the simulator (which seems to successfully simulate the card behaviour) is as follows. Firstly it provides an indication of the differences in execution times.

Secondly, it shows that the only reason that the log file download protocol does not operate on the card with large log files is the limited memory space.

Timing results for when the protocol was executed in the card are provided in table 2 (note that the results in the 'Estimated time' column are explained below). When the protocol is executed in the card, we tend to get slightly increased values, as was expected. This shows that the simulator does not correctly simulate the card processing time.

Table 2 GemXpresso Java Card results using log file of 40 bytes and estimated timings with variable Hash and MAC functions

	GemXpresso Card		
Part name	*Time (ms)*	*Std.Deviation*	*Estimated Time (ms)*
Connect	2689	24.76	2689
Validation	371	6.13	371
Send Log File	1091	5.27	$824 + x + y$
Verify Reply	1862	9.81	$1584 + 2x$
Disconnect	130	4.59	130
Total	6143	38.57	$5598 + 3x + y$

The Connect time is very big, compared with some results for the Open16K Java Card presented later. We could partially attribute this to the fact that, in this case, the client is implemented in Java, while in the other case it is implemented in Visual Basic.

To remove the influnece of the timings for the 'dummy' cryptographic functions, the execution times for the dummy hash and MAC over a 20-byte buffer were measured, and the results are given in table 3.

Table 3 Dummy hash and MAC execution times on the GemXpresso Card

Java Card	*Hash (ms)*	*MAC (ms)*
GemXpresso	128	139

The results in table 3 were used to estimate the time needed to execute the protocol on the GemXpresso Java Card, assuming that a hash takes x milliseconds and a MAC takes y milliseconds (see table 2).

5. CYBERFLEX OPEN 16K IMPLEMENTATION

In this section we present the implementation details when using the Cyberflex Open16K Java Card.

5.1 DESIGN AND LIMITATIONS

The software that accompanies the Cyberflex Open16K Java Card provides the developer with management utilities to download applets to the Java Card and offers a rather simple interface to test them.

As noted in §3.5, the same dummy cryptographic functions were used in both implementations. Because of constraints in the Java Cards, and to maintain consistency between implementations, the protocol was implemented using a 40-byte log file.

The Open16K JVM only supports data types of one and two bytes because the Java Card has an 8-bit microprocessor. Thus, the two-byte data type is implemented as two separate bytes. Also, an "int" (integer, 4 bytes) data type is not offered, which means that all results of arithmetic operations should be "type casted" or converted to the one-byte or two-byte data types.

For the Open16K Java Card, a "dump" card reader or card acceptance device (CAD) was available. Dump card readers are relatively unsophisticated, and synchronisation problems often arise, causing frequent problems with the experimental implementation, e.g. requiring PC reboots.

The simulator supplied for the Open16K Java Card did not operate correctly and a direct implementation to the Java Card had to be carried out. However, despite the difficulties, a stable implementation of the protocol was eventually produced.

5.2 IMPLEMENTATION

The functionality of the SendLogFileData and ReceiveALSSReply functions is described in §3.5 and depicted in figure 3. The PIN number is a 4-digit number and, once the user has been validated, she can invoke the rest of the functions.

The client side of the application was written in Visual Basic using a COM component provided with the Java Card that provided the interface with the PC/SC framework.

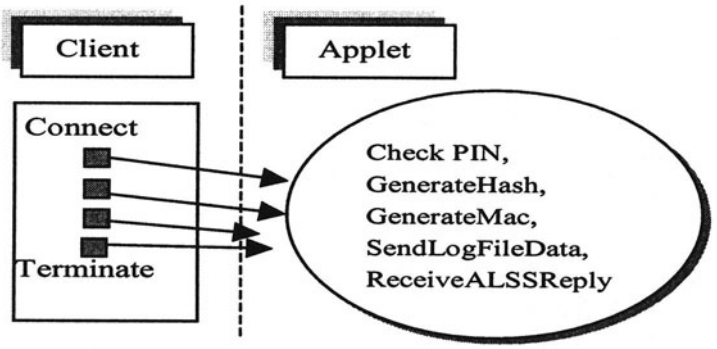

Figure 3 The Cyberflex implementation architecture.

5.3 RESULTS AND PERFORMANCE EVALUATION

The results of the timings are presented in table 4.

Table 4 Open 16K Java Card using log file size 40 bytes.

	Cyberflex Open16K		
Part name	*Time (ms)*	*Std.Deviation*	*Estimated Time (ms)*
Connect	280	23.66	280
Validation	220	15.81	220
Send Log File	2470	25.91	$2100 + x + y$
Verify Reply	1925	21.32	$1570 + 2x$
Disconnect	1210	20.98	1210
Total	6105	107.68	$5380 + 3x + y$

The connect time includes the initialisation time that is needed to access the COM component, to configure the connection with the card reader and select the applet. Once this is done, the Validation can take place. Afterwards, the client sends a request to the Java Card to receive the Log File. Finally, the client parses the response and answers to the card providing a hash that the latter has to verify. For each function we include host processing when applicable.

The Connect phase includes initialisation procedures carried out on the client, such as the Java Card's selection of which applet to run. The relatively small time is explained by the fact that much of the processing is done on the host PC.

The Validation process is very short, and involves comparing the given PIN with the correct one and informing the user whether it was correct.

The SendLogFile function is the most time-consuming function. It makes use of array copies as described in §3.5 to create the message sent to the client. Also, it computes a dummy hash and a dummy MAC. It then reads four files, totalling 74 bytes, from the file system of the Java Card. Finally, it returns 94 bytes of data to the client. The execution time of this function is higher than for the GemXpresso card. The only explanation would appear to be that reading and updating files in the Cyberflex card is slower.

The VerifyReply function accepts from 34 bytes the client, reads 40 bytes from within the Java Card and computes a hash. It then compares 26 bytes of data, advances a 32-bit sequence number by one using 8-bit arithmetic, computes one more hash, and writes two files, totalling 20 bytes, to the Java Card. Finally, it notifies the client whether the hash was verified.

The Disconnect function takes a relatively long time, compared with the Connect time. It can be assumed that this is PC/SC specific, or more precisely, it is related to the drivers that implement the support for the Open16K. Note that the Connect time is very short compared with the GemXpresso, although exactly the opposite is true with respect to the Disconnect time.

The time to execute the dummy hash and MAC over a 20-byte buffer were measured and the results are given in table 5.

Table 5 Dummy hash and MAC execution times on the Open 16K Card

Java Card	*Hash (ms)*	*MAC (ms)*
Open16K	177.5	192.5

Using the figures of table 5, we can estimate the protocol performance assuming that a hash takes x milliseconds and a MAC y milliseconds (see Table 4).

The communication speed between the Java Card and the client was 9.6Kbps. In the implementation, this amounts to 2.1% of the total communication time or to 126ms. If the communication speed was 57.6Kbps, the contribution of the data transfer delay would drop to 105ms. If there is need to optimize the protocol time to the level of hundreds of ms, then we would see fit to make the communication speed higher.

6. CONCLUSION

The log file download protocols were implemented on two different Java Card platforms. Although certain compromises had to be made with regard to the small log file size and the use of dummy cryptographic functions, we believe that we have shown that the concept of the secure log download protocol is a workable one.

The availability of a working simulator is very important in the development phases of the Java applet. It is not practical to have to download the applet every time to the Java Card in order to execute it. Also, it would be very useful for the simulator to be able to simulate aspects of the Java Card, such as memory restrictions and processor speed, using special options.

At a later stage, when cryptographic functions are provided as standard, it would be easy to replace the dummy functions.

The conclusion to be drawn from the speed of the hash and MAC functions [4] is that these dummy functions do not contribute substantially to the speed of the protocol. Other factors, such as file system access and internal array copying contribute to the speed slowdown.

Finally, from the overall execution speeds of the parts of the protocol, it is shown that a real-world implementation should be considered viable in the near future. Once the cryptographic functions become available and the Java Card performance is further enhanced (towards better handling of larger data structures), it is expected that more complex applications will become feasible.

7. ACKNOWLEDGEMENT

The authors would like to thank Chris Mitchell and Dieter Gollmann for their helpful comments.

References

[1] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Addison-Welsey Publishing Company Inc., 1994.

[2] Cyberflex. Cyberflex open16k reference manual. Technical report, Schlumberger, 1998.

[3] Gemplus. GemXpresso Reference Manual. Provided with the development kit, July 1998.

[4] Helena Handschuch and Pascal Paillier. Smart Card Cryptoprocessors for Public Key Cryptography. In *Third Smart Card Research*

and Advanced Application Conference Cardis'98, September 1998. to be published.

[5] MAOSCO. MULTOS Reference Manual Ver 1.2. `www.multos.com`, July 1998.

[6] Constantinos Markantonakis. Secure Log File Download Mechanisms for Smart Cards. In *Third Smart Card Research and Advanced Application Conference Cardis'98*, 1998. to be published.

[7] Constantinos Markantonakis. An architecture of Audit Logging in a Multi-application Smart card Environment. In *EICAR'99 E-Commerce and New Media Managing Safety and Malware Challenges Effectively*, 1999. Aalborg, Danemark.

[8] Sun Microsystems. Java Card 2.0 Language Subset and Virtual Machnine Specification. `http://www.javasoft.com/products/javacard/`, 1998.

[9] Sun Microsystems. Java Card 2.0 Programming Concepts. `http://www.javasoft.com/products/javacard/`, 1998.

[10] Sun Microsystems. The Java Card API Ver 2.0 specification. `http://www.javasoft.com/products/javacard/`, 1998.

[11] International Standard Organisation. *ISO/IEC 7816-4, Information technology - Identification cards - Integrated circuits(s) cards with contacts - Inderindustry Commands for Interchange.* International Organization for Standardisation, 1995.

[12] Schlumberger. Cyberflex open 16k. `http: www.cyberflex.austin.et.slb.com`, October 1998.

[13] OpenCard Framework Specification. Opencard framework. `www.opencard.org`, 1997.

[14] Jean-Jacques Vandewalle and Eric Vetillard. Developing Smart Card Based Applications Using Java Card. In *Third Smart Card Research and Advanced Application Conference - CARDIS'98*, September 1998. to be published.

[15] PC/SC Workgroup. Specifications for PC-ICC interoperability. `www.smartcardsys.com`, 1996.