Secure Audit Logging with Tamper-Resistant Hardware

Cheun N Chong Zhonghong Peng Pieter H Hartel

Dept. of Computer Science
University of Twente
The Netherlands

{chong,zhong,pieter}@cs.utwente.nl

Abstract

Secure perimeter schemes (e.g. DRM) and tracing traitor schemes (e.g. watermarking, audit logging) strive to mitigate the problems of content escaping the control of the rights holder. Secure audit logging records the user's actions on content and enables detection of some forms of tampering with the logs. We implement Schneier and Kelsey's secure audit logging protocol [1], strengthening the protocol by using tamper-resistant hardware (an iButton) in two ways: Firstly our implementation of the protocol works offline as well as online. Secondly, we use unforgeable time stamps to increase the possibilities of fraud detection. We provide a performance assessment of our implementation to show under which circumstances the protocol is practical to use.

1 Introduction

Digital content is so easily distributed, and dissociated from the meta-data that describes owner, terms and conditions of use etc. that copyright infringement is rife. Secure perimeter schemes such as digital rights management (DRM) alleviate the problem in some cases [2] but most (if not all) DRM systems are vulnerable to attacks. The raw content can then be redistributed, severely damaging the interests of the rights holder. Tracing traitor schemes trace leaks of content to the users who can be identified, and ultimately whose behaviour can be recorded as evidence. Many techniques exist to rediscover the identity and thence the rights on the content, such as cryptography, digital fingerprinting, watermarking etc. In this paper, we assume that users can be identified, and we concern ourselves with the issue of gathering information on the user's behaviour.

Secure audit logging records the actions of a user on an item of content and does so in a manner that allows some forms of tampering with the log to be detected. We implement Schneier and Kelsey secure audit logging protocol [1], using tamper-resistant hardware (TRH). For brevity in the sequel, we refer to Schneier and Kelsey [1] as "SK".

An audit log is an important tool to detect and to comprehend damages of a computer or network system caused by intrusions, defects or accidents. An audit log contains descriptions of noteworthy events. In our DRM experiment audit logs are generated in the user's personal computer (PC). The PC is a hostile environment (untrusted domain) because of its vulnerability against various malicious attacks. Therefore, the audit logs require protection to ensure its integrity.

SK renders audit logs impossible for an adversary to undetectably read, forge and delete. SK involves two parties: an untrusted machine and a trusted machine. The comment by SK that "the trusted machine may typically be thought of as a server in a secure location, or implemented in various ways, which includes a tamper-resistant token" has inspired us to use tamper-resistant hardware (TRH) as the trusted machine for secure logging. The TRH (or the trusted machine) we use is a Java iButton (*www.ibutton.com*) for the following reasons:

1. The iButton contains a programmable tamper-evident real time clock. The real time clock keeps time in $\frac{1}{256}$ second increments. This can be translated into seconds, minutes, days, months or years.

2. The iButton supports efficient implementations of common cryptographic algorithms such as SHA-1 (hashing), DES (symmetric encryption/decryption) and RSA (public key encryption/decryption and signature).

3. The iButton version 1.1 provides up to 6kB non-volatile RAM, the more expensive version 2.2 contains approximately 134kB non-volatile RAM.

We use the iButton as a trusted device to aid in the audit log creation in the manner proposed by SK. An iButton is too small to store a log of any useful size in a cost effective manner: A typical PC contains 40GB storage, i.e. around $300,000$ time more than the iButton.

Our DRM system has the usual Client/Server architecture. The Client is a user with her PC, which represents the untrusted domain. The Server is a trusted environment where content and license are stored. When the Client accesses the content piecemeal from the Server (e.g. by streaming), the latter is able to protect the content to some extent because the Client's actions can be monitored. However, when the Client downloads the content to the PC's non-volatile storage to accesses the content offline (i.e. disconnected from the Server), the Server is not able to monitor the Client's behaviour. We propose using secure audit logging with TRH to bring the security of offline DRM to the level of online DRM. The main contributions of this paper are:

1. To implement SK embedded in several auxiliary protocols and with the iButton.

2. To evaluate the performance of the implementation; thus investigating whether the iButton can be used effectively.

3. To strengthen SK by making sure that some of its security assumptions are valid by virtue of using the iButton. We generate core secrets and timestamps on the iButton instead of the untrusted PC.

To the best of our knowledge ours is the first attempt to implement SK and the first endeavour to analyse the performance of SK in general, and SK with iButton in particular.

A weakness of any system, which relies on TRH to coerce an untrusted Client into specific behaviour, is that the user may simply sever the connection between the Client and the TRH. We suggest a number of ways to discourage the Client from such behaviour:

- The Server is designed so that it insists on the iButton being present to authenticate and authorize the Client.

- Organizational policy (e.g. in a corporate intranet) is used to enforce the use of the iButton.

In both cases users are provided an incentive to maintain communication with the iButton: no iButton means no content.

The remainder of this paper is divided into sections as follows: Section 2 provides a user scenario. Section 3 describes related work. Section 4 explains SK using the iButton. Section 5 discusses concisely the refinement we have made of SK. Section 6 describes our implementation. Section 7 describes the feasibility of the implementation. Section 8 gives our performance analysis on the implementation. Section 9 concludes this paper also mentioning future work.

2 User Scenario

Figure 1 shows a user scenario where secure logging is needed. A Client (a user and her laptop) has downloaded a project proposal from the Server of her company. She needs to attach her iButton for downloading the proposal (for authentication purposes). She has also downloaded an associated digital license for reading this proposal on her laptop. The Server is able to monitor the Client's behaviour when the Client is online.

When the user carries her laptop out of her office, the Client is disconnected from the Server, and the Server has lost the ability to monitor the Client. The Client logs her behaviour, with the attached and valid iButton. Once she is back in her office, she reconnects to the Server (say for latest

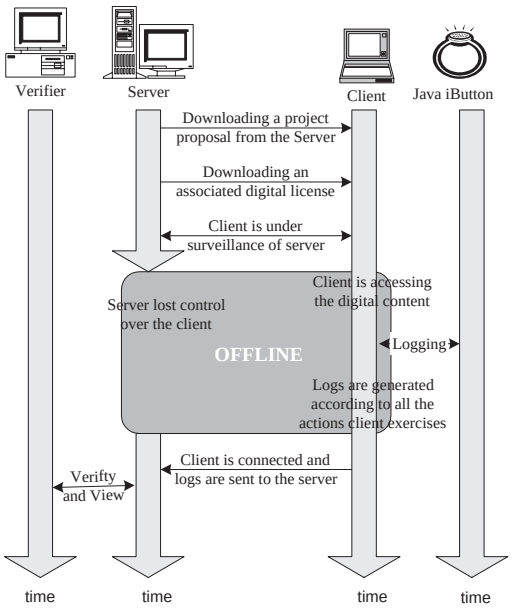

Figure 1: A user scenario of using iButton in a DRM application.

version of project reports) then all the logs stored at the Client are transmitted to the Server. A Verifier, which is a trusted Client, is able to view, validate and verify the log messages. The Verifier can be potentially the same system as the Server. The Verifier informs the Server either that the user has behaved correctly and that she is entitled to receive further content, or that something is wrong and that action should be taken.

3 Related Work

Ruffin [3] has produced a survey on audit logging. The definition of log he gives is:

> A log is an append-only write store and is a plain file where data are stored sequentially as they arrive.

Audit logs are generally used in an intrusion detection system (IDS), which is an automated system for detection of computer, network or database system intrusions. The audit logs of IDS can be used to trace the source of attacks thereby exposing the attacker's identity. Audit logs are also commonly used in e-commerce including DRM, not only as the common audit logs used in IDS, but also as audit trails to track the customers' activities. The audit logs can then provide the basis for charging the customer of using the services. Audit logs also offer a historical report on the customer's behaviour, which can be used for marketing purposes.

Shapiro and Vingralek [4] survey several mechanisms to manage the persistent state in a DRM system, including protected digital content, audit trail, content usage counts and decryption keys. One of the mechanisms they mention is secure audit logging. The two secure audit logging methods they cite are Bellare and Yee as well as Schneier and Kelsey.

Bellare and Yee (BY) [5] describe an audit log system that possesses the forward integrity property. "Forward integrity" is achieved by periodically altering the key to compute the MAC over the audit entries. BY alters the keys on a regular basis by feeding different secret values to a family of pseudo-random functions to generate the MACs over the entire log entries. If an adversary is able to compromise the current MAC key, it is unfeasible for her to deceive the historical entries generated. "Forward integrity" can be viewed as an integrity analogue of "forward secrecy"[6]. A protocol possesses the forward integrity property if a compromise of long-term keys does not compromise the past keys.

Schneier and Kelsey [1, 7] introduce a protocol for generating and verifying audit logs by using a linear hash chain to link the entire audit log entries so that some forms of tampering can be detected. The hash-chain is constructed by hashing each previous hash value of each log entry, concatenating with some other values.

SK provides a complete secure audit logging protocol, from the log creation to log verification and viewing. A trusted machine is needed during the log entry creation. The untrusted machine needs to communicate with the trusted machine from time to time to aid in audit logs creation and to send the audit logs generated. The logging process works while the connection to the trusted machine is available, and while the latter keeps the audit logs. On the other hand, BY generates the log entries without assistance from any external party. Similar to SK, the untrusted machine of BY main-

tains the audit logs.

SK and BY make illicit deletion of audit logs detectable. However, they are not able to protect the audit logs from deletion. Both SK and BY reckon that the deletion of log entries cannot be prevented by using cryptographic methods, but only by using write-only hardware such as CD-ROM, or paper printout. Additionally, SK and BY share another security weakness. If an adversary is able to compromise the untrusted machine at time t, i.e. obtains the key at time t, she is able to forge the log entry at time t.

In a later paper [8], Schneier and Kelsey improve their earlier idea by making audit log verification more efficient so that the system is more suitable for implementation in low-bandwidth environments [9]. They also present a signature format to simplify the task of designing strong protocols using security tokens. The security tokens use the padding bits, which are required for a signature algorithm (such as RSA) to pad a message before signing, as a channel to embed auditing information in the signed messages.

Instead of building special protocols to render audit logging secure, as proposed by SK and BY, it is also possible to use a trusted sub domain in an untrusted domain. Ordinary audit trails can then be secured by storing the logs inside the trusted sub domain provided the trusted sub domain offers sufficient storage capacity. There are two categories of mechanisms available: (1) Tamper-resistant software (TRS) and (2) Tamper-resistant hardware (TRH).

There are at least two ways of realizing TRS in an untrusted domain. The first is by using homomorphic encryption, whereby instead of calculating with raw data, (equivalent) calculations are applied to encrypted data [10]. The second realization of TRS is code obfuscation [11]. This latter technique consists of applying various standard program transformations such that the original program becomes difficult to analyse (by machine) and/or recognize (by people).

It is generally possible to transform programs in various different ways because of the generous amount of redundancy present in programs. However, it is not possible to apply just any transformation. Obfuscation by adding gratuitous pointer manipulation will stick out "like a sour thumb", which fails the stealth criterion of Collberg et al [12]. Opponents will easily identify such unstealthy transformations and attack them. Once an analysis technique for a particular obfuscation transformation has been found, it would be possible to break not just one key but also all the keys that depend on this transformation.

TRH adds a trusted subdomain in the untrusted domain by using add-on hardware to the PC. A smart card (and Java card, which is an instance of a smart card) is a general hardware token that is used to store holders' sensitive information, such as bank account data, PIN numbers, passwords, private keys and other secret data. The Java iButton (*www.ibutton.com*) is a relatively high security example of TRH. Many applications [13, 14, 15, 16, 17] are developed using the TRH, for their tamper-resistant features. As far as we know, we are the first to implement the secure audit logging using the iButton.

Cryptographic coprocessors, such as the IBM 4758 PCI SecureWay Cryptographic Coprocessor (*http://www-3.ibm.com/security/cryptocards/*) offer more security than a smart card. The IBM coprocessor is able to add a high-security environment to a variety of operating systems on a PC and provide a tamper-resistant environment for storing keys and performing sensitive processing. Dyer et al [18] have run a case study on such coprocessor, examining the performance of this product as a system. Gutmann [19] examines various concerns in designing the coprocessor, and explores alternatives for improving the performance of crypto operations on the hardware.

Weis and Lucks [20] explore the speed of modern block ciphers implemented in Java. They provide relevant benchmarks on an Intel Pentium 200 MHz MMX CPU machine, but unfortunately not on a smartcard/iButton. Schneier and Whiting [21] measure the performance of various AES candidates, e.g. Twofish and Rijndael on a smartcard with an architecture comparable to our iButton. Their measurements are similar to our iButton measurements for DES. Durand [22] measure a $40\times$ better performance of RSA than we do, which is at least partly due to their using a 32-bit 40MHz smartcard. Our iButton is only an 8-bit CPU running at 10MHz. We interpret Durands measurements as an indication that the performance of iButtons could be significantly improved in future,

thus offering better performance also to a secure audit logging application.

4 The Protocols

Figure 2 illustrates the protocols in our secure audit logging method. SK1 and SK2 are from SK. The others, AP, P1 and P2 are our own protocols.

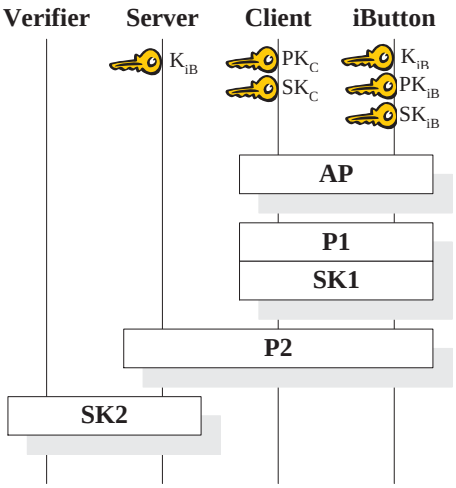

Figure 2: Overview of the secure audit logging method.

The four protocols shown in the figure are:

1. SK1: This is Schneier and Kelsey's protocol for creating and closing audit logs. SK1 runs between the Client and the iButton. We discuss SK1 in section 4.1.

2. AP: The AP protocol is a standard smartcard/iButton authentication protocol. AP authenticates the Client to the iButton. This is to guarantee that the iButton talks to the legitimate Client. AP runs between the Client and the iButton. We explain AP in section 4.2.

3. P1: The P1 protocol is for the iButton to generate the authentication keys and timestamps for SK1. Similar to AP, P1 runs between the Client and the iButton. We explain P1 in more detail in section 4.3.

4. P2: The P2 protocol sends the audit logs maintained at the Client to the Server and synchronizes the iButton real-time clock with the Server clock. The protocol also enables transmitting securely the initial authentication key stored on the iButton to the Server. Therefore, P2 involves the Client, the iButton and the Server. We elaborate P2 in section 4.4.

5. SK2: The SK2 protocol is for verifying and viewing the audit logs stored in the Server. We describe SK2 in section 4.5.

As shown in Figure 2, the Client and the iButton own different sets of key pairs. PK_C and SK_C are the public key and private key of the Client, respectively. PK_{iB} and SK_{iB} are the public key and private key of the iButton, respectively. The iButton and the Server share a secret key, K_{iB}. The public/private keys and the shared secret key are preloaded on the iButton before it is deployed.

4.1 SK1

SK1 creates and closes an audit log. Section 4.1.1 explains the assumptions we have made for our implementation. Section 4.1.2 lists the notations and symbols we use to describe the protocols. Section 4.1.3 elaborates the rules for constructing the audit logs.

Sections 4.1.2 – 4.1.3 are taken from SK and reproduced here in abbreviated form to make our paper self contained. We focus on the technical details of the protocol and refer the reader to Reference [1] for the motivations behind the protocol.

4.1.1 Assumptions

We have made some assumptions that we believe to be reasonable while implementing the protocols. The main reason is to facilitate the implementation of SK in the resource constrained iButton.

1. The connection between the Client and the Server, as well as the Verifier and the Server is established by using the Secure Socket Layer (SSL) mechanism.

2. Without restricting generality, there is only *one* Verifier, *one* Server, *one* Client and *one*

iButton in the audit logging process. There is only *one* audit log file created from the process.

3. If the iButton is removed from the iButton reader halfway through an instruction then the log file is closed abnormally with a reason stated in the log (see Figure 5).

From SK (begin)

4.1.2 Notation

For better understanding of the remainder of this paper, we use the following notations from SK.

- ID_x represents a unique identifier string for an entity x.
- $PKE_{PK_x}(K)$ is the public-key encryption of K using the public key, PK_x of x. The corresponding private-key decryption of K is $PKD_{SK_x}(K)$, where SK_x is x's private key (secret key). We use the RSA algorithm because it is available in the iButton version release 1.1.
- $SIGN_{SK_x}(Z)$ is the digital signature of Z, under x's private key. We use RSA with a key size of 128 bits.
- $E_K(X)$ is the symmetric key encryption of X under key K. D_K is the corresponding symmetric key decryption, which we use in addition to SK notations. We have renamed K_0 in the SK to RK_0 to distinguish it from the log entry encryption key in section 4.1.3. K can be RK_n, (a random session key) or K_{iB}. We use single DES.
- $MAC_{K_0}(X)$ is the message authentication code, under the symmetric key K_0, of X. We use a one-way hash MAC algorithm, namely HMAC.
- $hash(X)$ is an one-way collision-resistant hash function for which we use the SHA1 algorithm.
- X, Y represents the concatenation of X with Y.

4.1.3 Log Entry Construction Rules

We now summarize the SK log entry construction rules below, indicating the role of the iButton where appropriate.

1. D_j is the real log data in jth log entry of ID_{log}. For a DRM application, the data should encapsulate information from the digital license, actions the Client exercised on the digital content, behaviours of the content renderer and other relevant information that occurs at the Client.
2. W_j is the log entry type of the jth log entry.
3. A_j is the authentication key for the jth entry in the log. This is the core security of SK.
4. $K_j = hash(''EncryptionKey'', W_j, A_j)$ calculates the jth log entry symmetric encryption key.
5. $Y_j = hash(Y_{j-1}, E_{K_j}(D_j), W_j)$ is the chained hash. Each log entry contains an element in a hash chain to authenticate the values of previous log entries [23].
6. $Z_j = MAC_{A_j}(Y_j)$ is the MAC code of Y_j generated under jth authentication key, A_j.
7. $L_j = W_j, E_{K_j}(D_j), Y_j, Z_j$, is the jth log entry consisting of the concatenation of the four values listed.
8. $A_{j+1} = hash(''IncrementHash'', A_j)$ generates a subsequent authentication key from the previous one.

4.1.4 The Process

SK1 is divided into two parts: (1) create log and (2) close log. We pay most attention to the audit log creation process because we are interested to know the influence of the iButton on the Client's performance.

Create Log Figures 3 and 4 show the process of creating an audit log. In SK, the Untrusted machine (the Client) sends its public key certificate to the Trusted machine (the Server) for client authentication. The iButton, which we use as a

Trusted machine as explained in section 1, authenticates the Client using a standard authentication method discussed briefly in section 4.2.

The steps and messages of Figure 3 and Figure 4 can be explained as follows:

1. To start audit logging, the Client takes the initiative by activating P1. After P1, the Client has the initial authentication key, A_0 and the current encrypted timestamp, $E_{K_{iB}}(d_0)$. SK leaves it to the Client to decide A_0 and d. We leave these two items generated on the iButton, as explained in section 4.3.

2. The Client forms a random session key, RK_0; the timeout period that the Client will wait for the response from the iButton, d_0+; a unique identifier for this log file, ID_{log}; and a nonce known as step identifier, p. The Client concatenates p, $E_{K_{iB}}(d_0)$, and A_0 to form the message X_0.

3. The Client encrypts RK_0 with the iButton's public key, PK_{iB} and encrypts the signed X_0 with RK_0. The Client forms a message, M_0 by concatenating p, ID_C, the encrypted RK_0, and the encrypted signed X_0.

4. The Client generates and stores the hash of X_0 for validation in the subsequent steps.

5. The Client sends the message, M_0 to the iButton.

6. The iButton retrieves the random session key, RK_0 by doing private key decryption. The iButton can then decrypt and retrieve X_0 by using RK_0 as a decryption key. The iButton validates X_0 by verifying the signature. Finally, a new random session key, RK_1 is generated.

7. The iButton forms message X_1, by concatenating the step identifier p, ID_{log}, and hash of X_0. The iButton generates message M_1 by concatenating ID_{iB}, encrypted RK_1 with the Client's public key, PK_C, and the encrypted signed X_1 with RK_1.

8. The iButton sends the reply message, M_1 to the Client.

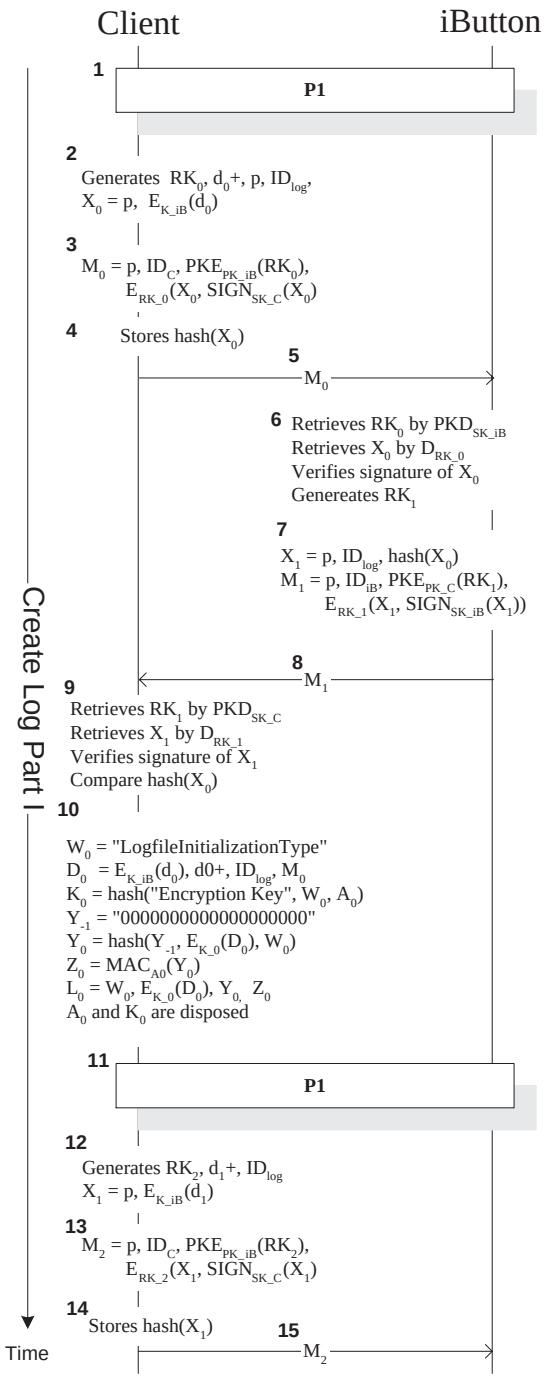

Figure 3: The protocol of creating audit logs (steps 1 to 15).

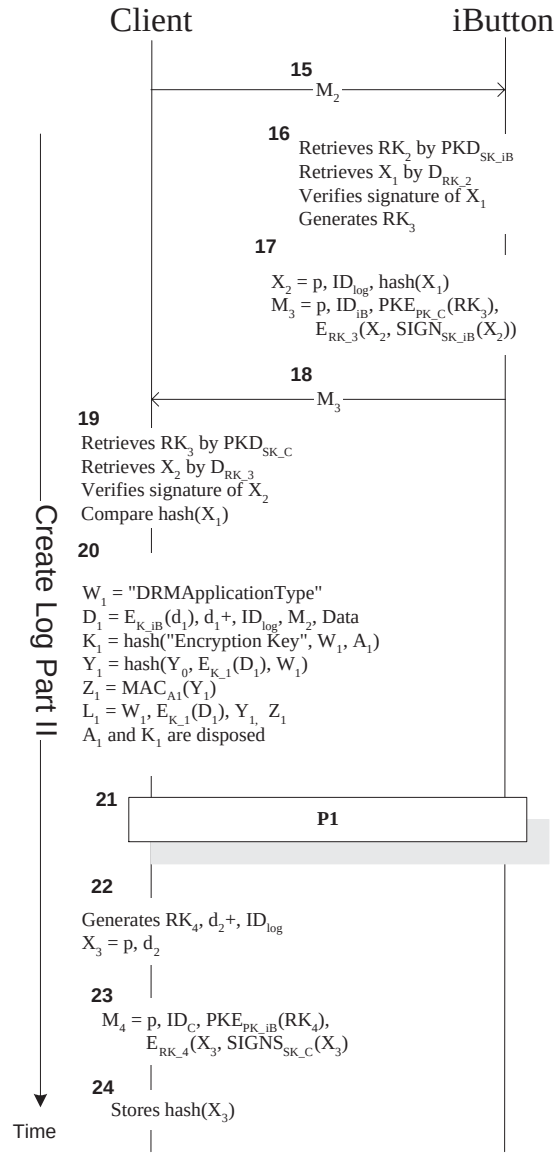

Figure 4: The protocol of creating audit logs (steps 15 onwards).

9. When the Client receives M_1 from the iButton, the Client verifies the message. The Client compares the hash value of X_0 from the iButton with the stored hash value.

10. The Client generates the first log entry. The first log (of type, $W_0 =''$ $LogFileInitializationType''$) must be properly formed at the Client or else the Server will suspect that the Client has been tampered with. The first data field, D_0 encapsulates $E_{K_{iB}}(d_0)$, d_0+, ID_{log}, and M_0. The Client generates the first log entry encryption key, K_0. There is no previous hash value to start the hash chain since this is the first log entry. Therefore, we set the initial hash value, Y_{-1} to an array of zeros. Hash value, Y_0, and the MAC value, Z_0 are generated. A_0 and K_0 are disposed.

11. Similar to step 1, to start audit logging, the Client enables P1 and obtains a new authentication key, A_1 and a new encrypted timestamp, $E_{K_{iB}}(d_1)$.

12. The Client generates a new random session key, RK_2, a new timeout, d_1+. The ID_{log} is the same ID_{log} generated at step 1. The Client concatenates p and $E_{K_{iB}}(d_1)$ and forms X_1.

13. Similar to step 3, the Client produces message M_2.

14. Similar to step 4, the Client stores the hash value of X_1 for future validation.

15. The Client sends message M_2 to the iButton.

16. The iButton repeats step 6.

17. The iButton repeats step 7.

18. The iButton sends the generated message, M_3 back to the Client.

19. The Client repeats step 9 for verification on the messages received from the iButton.

20. The Client repeats the procedures of generating the first log entry (step 10) to generate subsequent log entries, with $W_j =''$ $DRMApplicationType''$, where j is the log entry number. The only difference is that at

this stage, the Client is able to generate a hash value of previous log entry, Y_1 to form the hash chain.

The procedures from step 21 onwards, as can be seen in Figure 4 are repeated for generating new log entry of type $''DRM\,ApplicationType''$ for DRM application. The Client creates an log entry (step 10 and step 20) conforming to the construction rules discussed in section 4.1.3.

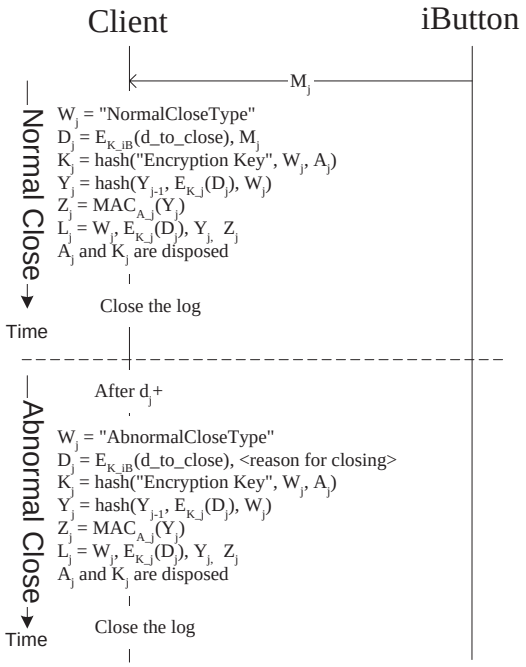

Figure 5: The SK protocol of closing a log file.

Close Log When the Client stops the DRM application, the log file must be closed. There are two types of closing: a normal closing type and an abnormal closing type as can be seen in Figure 5.

When the Client receives the last response message, M_j from the iButton, the Client records the log file closing time. The Client then generates the final log entry with type of $''NormalCloseType''$. However, if the Client has not received the response message from the iButton after a timeout, d_j+, the Client closes the log file with $''AbnormalCloseType''$ and states the possible reasons in the log entry data.

If the iButton is detached from the Client during the SK1 log creation, we assume that the audit log will be closed under $''AbnormalCloseType''$. This will be implemented in the near future.

From SK (end)

4.2 AP

AP is a standard smartcard/iButton authentication protocol. It authenticates the Client to the iButton to ensure that the Client is genuine. The details of the protocol are beyond the scope of this paper.

4.3 P1

P1 generates the authentication keys, A_j and to generate the timestamp, d_j for recording the log entry. We have refined SK1 by deciding that the iButton should generate the keys and timestamps, as can be seen in Figure 6. We improve SK1 to the extent that A_0, which represents the core security of SK1, and the timestamps are generated in a trusted subdomain. We encrypt the timestamp generated using the key, K_{iB}, which is shared between the iButton and the Server. By doing so time stamps cannot be manufacutured by the client (who does not have access to K_{iB}).

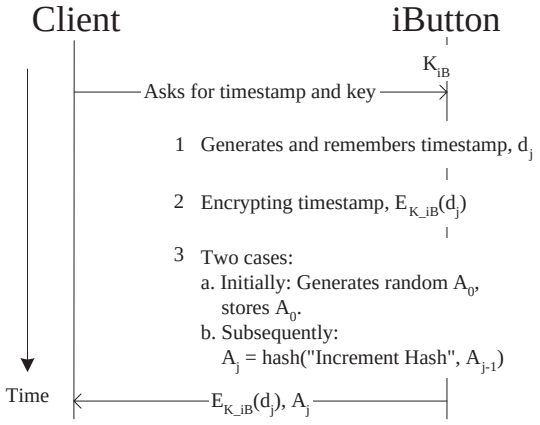

Figure 6: The P1 protocol for generating the initial authentication key A_0 and timestamp d from the iButton.

The Client first requests an encrypted timestamp and an authentication key from the iButton for the current log entry. The iButton generates the timestamp using its real-time clock and encrypts the time stamp with the iButton secret key, K_{iB}. The iButton then remembers the first timestamp, i.e. the timestamp for the initialisation log entry, with type $''LogFileInitializationType''$ and also the last timestamp, for the close log entry, with type $''NormalCloseType''$ or type $''AbnormalCloseType''$. We will come back to this in section 5.;

The iButton then generates a random key, as the initial authentication key, A_0 if it is the first log entry the Client constructs; otherwise, the iButton hashes the previously existing authentication key, A_{j-1} concatenated with a message to generate the next authentication key, A_j. The iButton stores the initial authentication key, A_0 and current authentication key, A_j for generating subsequent keys, A_{j+1}. After finishing the generations, the iButton sends the encrypted timestamp and the authentication key back to the Client.

4.4 P2

P2 synchronizes the iButton clock to the Server clock, which is a trusted clock. P2 also sends the log file maintained at the Client and the corresponding initial authentication key on the iButton to the Server, as shown in Figure 7. The Server and the iButton share a secret symmetric key, K_{iB}, which is used to encrypt the data exchanges between the iButton and the Server (and also the timestamps) from the Client.

The Server encrypts its current time with K_{iB}, and sends the encrypted time to the iButton for time synchronization. The iButton then decrypts the message and adjusts its real time according to the received time. Once the time is synchronized, the iButton sends back the accepted encrypted time to the Server as an indication that the time is synchronized.

After the clock synchronization, the Server sends a request message to the Client asking for the available log file. The Client forwards the request to the iButton. The iButton sends the encrypted A_0, as well as initialisation timestamp, d_0 and close timestamp, d_j to the Server via the Client. At the same time, the Client sends the

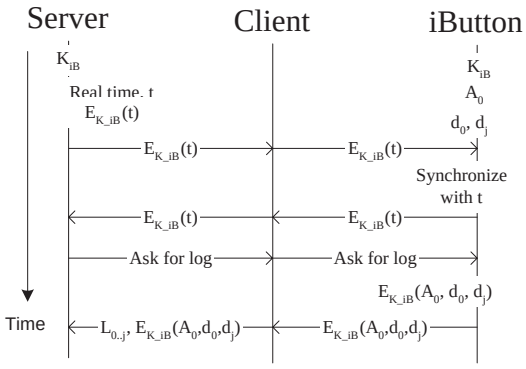

Figure 7: The P2 protocol of synchronizing the iButton real-time clock and sending audit logs to the Server.

available log file (corresponds to the ID_{log} of A_0) to the Server. In the DRM system, P2 is transparent to the Client.

From SK (begin)

4.5 SK2

SK2 verifies and displays the audit logs to the Verifier. We have changed slightly the SK Verifier in that it does not store the Client's log file locally. The Verifier reads and verifies the log file remotely from the Server. In other words, the log file is stored securely in the Server, and the cryptographic processes are operated at the Server.

Figure 8 illustrates the process. The Verifier forms $Q[0..n]$ encapsulating the log entry numbers, 0 to n of log file ID_{log}, concatenating with the entry numbers the corresponding log entry type, W. The Verifier sends the generated message, $M_1 = p, ID_{log}, Q[0..n]$ to the Server.

The Server then proceeds to verify log entries, as shown in Figure 9. The Server retrieves the initial authentication key, A_0 (or generates subsequent authentication key, A_n from A_0). The Server validates the first log entry, L_0 by validating the hash value, Y_0 and the corresponding MAC value, Z_0.

If the hashes and MACs validation pass, the Server retrieves the data, D_0 from the log entry and the Verifier is able to read the data. The Server decrypts the timestamps using the key it

shares with the iButton. To examine whether the audit logs have been manipulated, the initialisation and close timestamps are compared with the trusted ones the Server receives from the iButton. The Server repeats the process until the last log entry requested, n is reached.

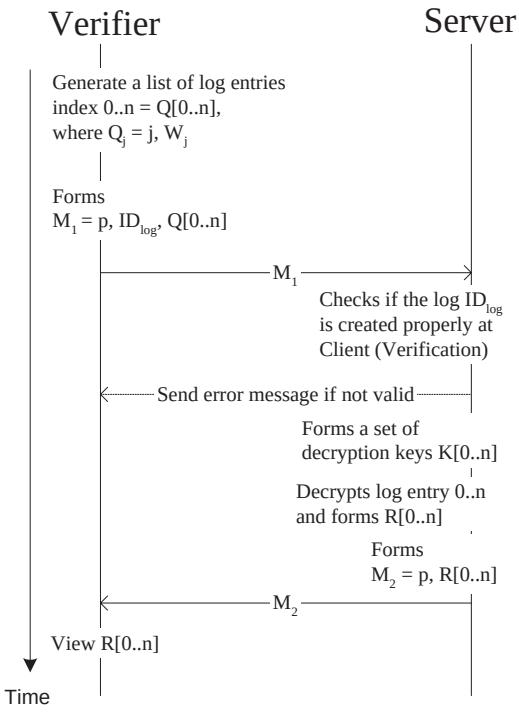

Figure 8: The protocol of verifying and showing audit logs at the Server.

From SK (end)

5 SK Refinement

We have refined SK by introducing two auxiliary protocols, P1 and P2. P1 lets the trusted iButton instead of the untrusted client generate the authentication keys and the timestamps. The iButton remembers the timestamps for initialization and closing of the log. Additionally, the Client only possesses the encrypted timestamp from the iButton for audit logging, i.e. the integrity and confidentiality of the timestamps are ensured. P2

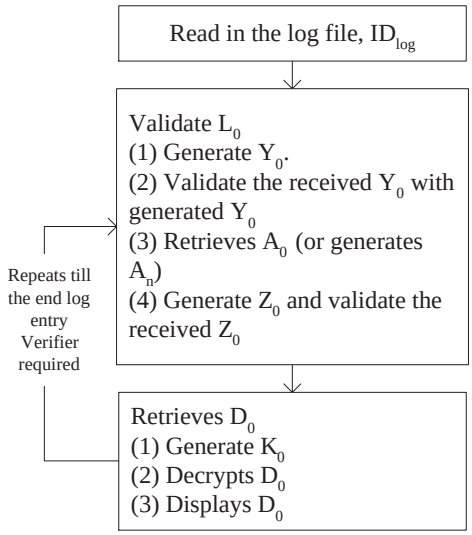

Figure 9: The process Server validating the log entries.

enables the iButton to transfer the encrypted initial authentication key and stored timestamps using the shared secret key with the Server. In other words, P2 is able to guarantee the integrity of the initial authentication key and the timestamps during the transmitting process.

As pointed out by Schneier and Kelsey [1, 7], there is a security weakness in SK. If an adversary is able to compromise the Client by getting hold of the key A_t, directly after it has been generated at time t, the adversary is able to falsify the log entry at time t. The adversary is also able to create more counterfeit log entries and remove some log entries. The Verifier is not able to detect the frauds because the adversary is able to construct another "truthful" hash-chain and MAC over the entire log entries using the compromised authentication key.

In our refinement of SK, by using encrypted timestamps, we are able to detect some of the aforementioned frauds. If the adversary wishes to create more log entries, she has to obtain the cooperation of the iButton to generate valid timestamps. The adversary does not have the right key, so she cannot fabricate arbitrary timestamps herself. The adversary can reuse genuine encrypted timestamps, but the verifier will notice missing or duplicate time stamps, or time stamps that are

presented out of order. The adversary will also be caught truncating the log file when the time of truncation does not match the time of the last transaction remembered by the iButton. In case of tampering, the user can be held responsible for the entire period between the log initialization time and close time.

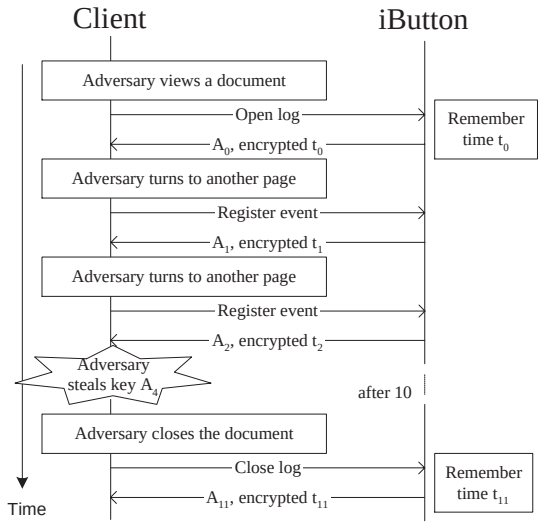

Figure 10: An adversary views a protected document and steals the key at time $t = 4$, during the logging process.

We now present a concrete example of the difference between the original SK and our version as shown in Figure 10. An adversary, who owns a protected document and an associated license, starts the content renderer on the Client. The first log entry, initialization log, is then generated. Subsequent log entries are generated when she turns to other pages of the document. At time t_4, the adversary successfully steals the key A_4. She does not stop browsing the document, but keeps reading until time t_{10}. She closes the document at time t_{11} and so the log file is closed. She wants to deny the fact that she has viewed the document from t_4 till t_{10}. Instead the adversary wants the Verifier to believe that she has viewed the document until t_3. The adversary would wish to do so for example when she is charged on a per time unit basis. The adversary thus removes the log entries from t_4 onwards, and creates a false close log entry using timestamp t_4. As she possesses the key A_4, she is able to construct a valid hash-chain and forge the MACs for this fraudulent log. In our version of SK, the Verifier is able to detect the forgery because the iButton remembers the log closing time (t_{11}), which in the scenario above does not match t_4. The original SK protocol does not detect this situation.

Using encrypted time stamps improves the security of the protocol but weaknesses remain. For example if after a perfectly legitimate run of the protocols the user starts viewing the content using an application that does not log any actions, then this will not be noticed.

6 Implementation

We use iButton version 1.1 in our implementation and a Blue Dot Receptor (with a Serial Port Adapter) for attaching the iButton to the Client machine. We have implemented the Client in Java. The Verifier and the Server are implemented as Java applets and servlets. We use the Java cryptography extension library provided by Institute for Applied Information Processing and Communications (*jcewww.iaik.tu-graz.ac.at*) to implement cryptographic operations, i.e. encryption, hashing and signature.

We have implemented the iButton code using the iB-IDE API provided by Dallas Semiconductor, Corp (*www.ibutton.com*), which is compliant with OpenCard Framework (*www.opencard.org*) and Java Card API (*java.sun.com/products/javacard/*).

We have implemented a complete logging system from the Verifier to the iButton using the protocols discussed in section 4. The Client application we have created needs the iButton to run the logging operation. A status window of the Client application displays the status of the logging process. A log file is created encapsulating the encrypted log entries.

We use Apache/Tomcat (*jakarta.apache.org*) on a Pentium III 850MHz 256MB RAM machine as the Server. To upload the log file, the Client is connected to the Server via Internet Explorer. The Server waits for the Client to present the iButton before any further processing. The log file and the initial authentication key are uploaded once a

valid iButton is detected by the Server. The key is stored encrypted by the shared key between the Server and the iButton in a file.

The Verifier also interfaces with the Server via the Internet Explorer. The Verifier enters the entry numbers and the ID of the log file she wishes to read. The verification process is activated at the Server. If the verification fails, e.g. due to some missing log entries, or unauthorized modification on the log entries etc., the process halts and displays an error message. Otherwise, the data of the log entries are shown to the Verifier.

7 Discussion

In this section, we describe the feasibility of using the iButton, describing the problems we have encountered during our implementation. The iButton has several good features as listed in section 1. Unfortunately, it also poses some disadvantages.

Limitation of RAM : iButton version 1.1 only contains up to 6kB fast non-volatile RAM. However, roughly 2kB out of the 6kB is consumed by the system storage demands, such as memory manager, APDU [1] communication layers and the Java virtual machine *per se*. We have to use the limited approximately 4kB RAM to implement the cryptographic operations, store the iButton's keys pair, keep the Client's public key and other items, which are listed in Table 1.

We have to store the public/private key pairs in exponents and modulus form (in bytes) in the iButton version 1.1. The length of the private key exponent and the modulus are 128 bytes. The public key exponent is only 3 bytes, because we use the Java API to generate key pairs and we convert the key pairs to the Java API big integer before we change them to bytes [24]. The iButton version 2.2 provides a more space to store the keys.

The average length of messages exchanged buffer between the Client and the iButton, as shown in Figure 3 and Figure 4 is approximately 128 bytes. To solve the memory problem, we have tried to shrink the size of the iButton applet code by making several optimisations, e.g. reusing the space for allocating different variables.

[1] APDU (Application Protocol Data Units) is a standard protocol the token communicates with the Client.

Items	Size (bytes)
iButton applet code size	1996
iButton's public exponent	3
iButton's private exponent	128
iButton's modulus	128
Client's public exponent	3
Client's modulus	128
Authentication key, A_0	8
Message exchange buffer	128
Total	2522

Table 1: Size (bytes) of items that consume the RAM memory of iButton version 1.1 in our implementation.

Unstable connection : The connection between the iButton and the Client is not robust; the connection is frequently terminated unexpectedly during the audit logging process. We have tried to reduce this problem by re-configuring settings of the iButton, e.g. connection time out between the iButton and the Client.

8 Performance Analysis

As we have mentioned earlier in section 4.1, we are interested to know how the iButton affects the performance of the Client during audit logging. This allows us to determine if the iButton is practical for secure audit logging in a DRM system. We have run several performance tests on the implementation. We have used a PC machine Pentium III 800MHz with 256MB RAM as the Client machine for performing the tests.

We measure the time for generating a number of log entries on the Client, as discussed in section 8.1. We then run the experiments on the iButton in section 8.2 to measure the time needed for various cryptographic operations.

8.1 Testing on the Client

We have measured the time for creating different numbers of log entries, from 1 to 10 on the Client, as shown in Figure 11. We can see from the figure that the graph is nearly a straight line. Generating 1 log entry takes approximately 1 minute.

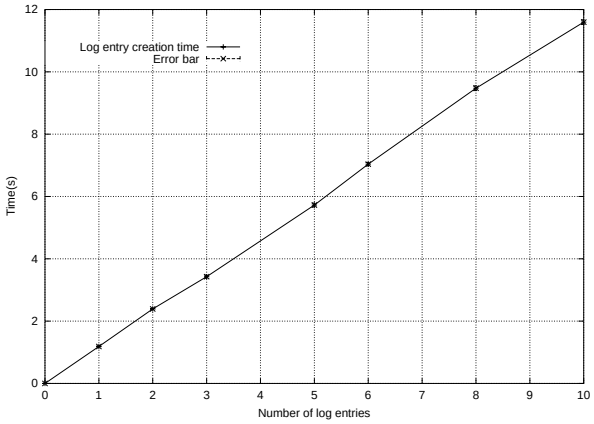

Figure 11: The time measurement for generating 1 to 10 log entries.

To explore why it takes roughly 1 minute to generate only 1 log entry, we have measured the time spent for performing cryptographic operations on the iButton, as explained in section 8.2. We believe that the cryptographic operations are the main causes to the long time taken for generating log entries. The cryptographic operations include:

1. DES encryption and decryption, with 64-bit key.
2. RSA encryption and decryption, with 128-bit keys.
3. SHA1 hash generation
4. Signature generation and verification with RSA and SHA1 algorithm.

8.2 Testing on the iButton

Figure 12 illustrates our testing procedure for measuring time on the iButton. We read the start time on the iButton right before the iButton starts the calculation under investigation. We read the stop time on the iButton once it stops the process.

The time taken is the value of deducting the start time from the stop time. The result is then transmitted back to the Client. As only time with seconds-precision is available on the iButton (the software does not provide access to the $\frac{1}{256}$ second accuracy of the hardware clock), we have run the

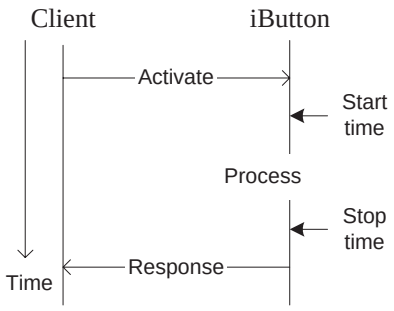

Figure 12: The testing procedure we have run to measure the time of each cryptographic operation.

process repeatedly on the iButton, dividing the time measures by the number of repetition. We take the average value of 20 measurements as our final value using the standard deviation as error margin.

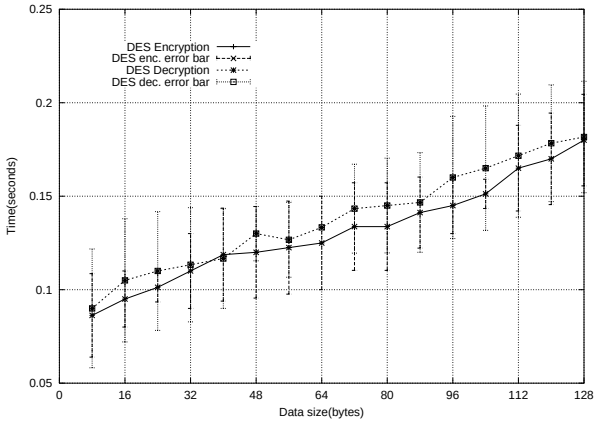

Figure 13: Time needed to encrypt and to decrypt a message of different size (bytes) on the iButton with DES algorithm 64-bit key.

Figure 13 shows the time spent for encrypting/decrypting a message of various size in bytes (from 8 to 128) using the 64-bit key DES algorithm on the iButton. The encryption and the decryption times are around 200 ms for large message (size bigger than 128 bytes).

Figure 14 shows the time consumed for encrypting a message of sizes from 8 bytes to 128 bytes, using 128-bit public key of RSA algorithm; and de-

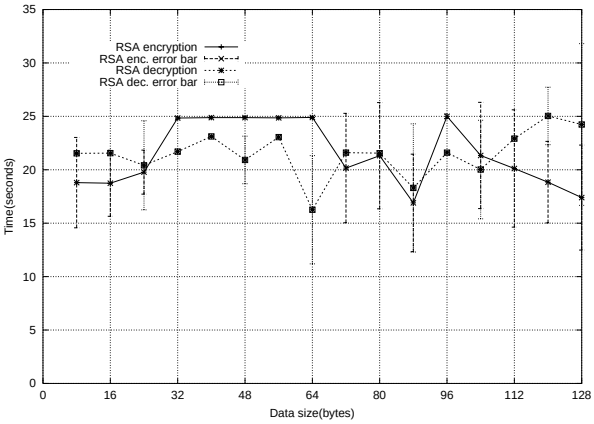

Figure 14: Time needed to encrypt a message of different size (bytes) and decrypt the message on the iButton with RSA algorithm 128-bit key.

crypt using the corresponding 128-bit private key on the iButton. As can be observed in the figure, the graph is nearly a straight line when considering the error margin. RSA encryption takes about 25 seconds, while 22 seconds are needed for RSA decryption.

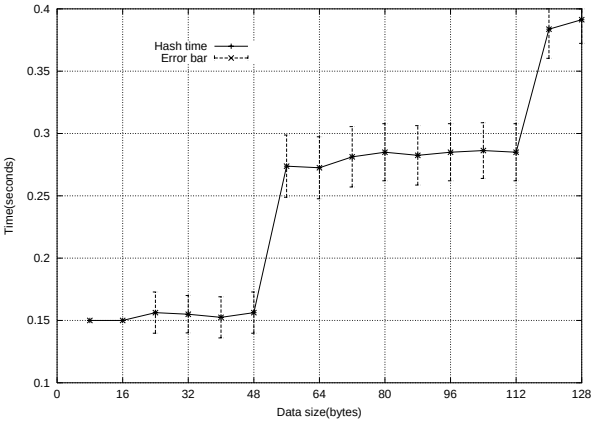

Figure 15: Time needed to hash a message of different size (bytes) on the iButton.

Figure 15 illustrates the time spent for hashing a message of sizes range from 8 to 128 bytes using the SHA1 message digest algorithm. As shown in the figure, the time spent for hashing 56 bytes is almost double the time spent for hashing 48 bytes. This is due to the message padding to 64 bytes [25].

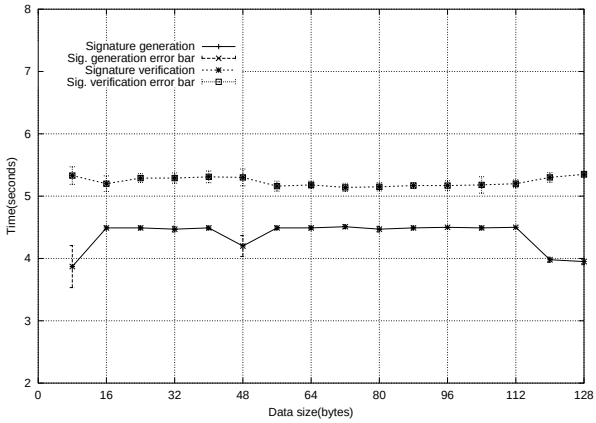

Figure 16: Time needed to sign a message of different size (bytes) and verify the signature on the iButton.

Figure 16 shows the time needed to sign a message using SHA1 and RSA and to verify the signed message. As can be seen in the figure, it takes 4 to 5 seconds to sign a message on the iButton, but it takes 5 to 6 seconds to verify the signature.

We do a back of the envelope calculation referring to Figure 3 to confirm the measurement on the log entry generation time we obtain. The iButton takes less than 1 second for generating the timestamp and authentication key, i.e. to complete the protocol P1. The cryptographic operations on the iButton, as mentioned earlier consume most time. RSA private key decryption and public key encryption take approximately 24 seconds and 20 seconds, respectively. DES encryption and decryption need about 0.1 second, respectively. Signature generation and verification require roughly 5 seconds and 4 seconds respectively. These values are taken according to the size of the message we have done early.

Each cryptographic operation performed on the Client takes less than 1 second, including public key cryptography. We measured that to transmit message from the Client to the iButton and vice versa takes about 1 ± 0.02 second for messages of sizes bigger than 128 bytes. Other operations, including the key generation and log entry generation on the Client only take approximately 1 to 3 seconds in total. All of these values aggregate to

roughly 1 minute, as expected.

To conclude our performance analysis, as it takes about 1 minute just to generate 1 log entry, the iButton is not practical to be used in a system that requires frequent logging. However, if the system only logs the main events, such as playing a 4-minute song, reading an eBook, and other content access taking a longer time, we believe that the iButton is practical. Note that in our system logging overlaps with the actual content rendering.

For logging frequent events, we believe that we could use iButton as a bootstrap device for ensuring the trustworthiness of the first audit log entry, and we could do the subsequent log entries creation for frequent events without the presence of the iButton. This issue remains open for future investigation.

9 Conclusions and Future Work

We propose using secure audit logging in a DRM system where the Client is not permanently online. This allows the Server to obtain knowledge of the Client's behaviour during offline periods. We implement Schneier and Kelsey (SK) secure audit logging protocol, using tamper-resistant hardware (an iButton) as the SK trusted machine. Our main contributions are:

- To implement SK embedded in several auxiliary protocols with the iButton.
- To report on the performance of the audit logging with the iButton.
- To refine and strengthen the SK log creation protocol, by generating the core secrets and timestamps on the iButton.

We discuss the problems we encounter during the implementation: the limitation of RAM of the iButton and the unsteady connection between the iButton and the Client.

The performance evaluation reveals that it takes about 1 minute for generating 1 log entry. We reckon that this is not practical for a system that requires frequent logging but feasible for a system that only needs to log the main events such as playing a 4-minute song. To make the iButton implementation also practical for recording more frequent events in future work we intend to use the current implementation recursively: one entire log on the untrusted PC would then correspond to one log entry that involves the iButton. The performance could also be improved dramatically using a more powerful iButton.

The main problem with all secure audit logging protocols is that if a log entry at time t is compromised, then none of the log entries after time t can be trusted. Our use of securely encrypted time stamps can solve this problem in some (but not all) cases. We believe that we can improve the security further by asking the iButton to do a little more work. This remains open for future work.

References

[1] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines," in *The 7th USENIX Security Symposium Proceedings*, pp. 53–62, USENIX Press, January 1998.

[2] C. N. Chong, R. van Buuren, P. H. Hartel, and G. Kleinhuis, "Security attribute based digital rights management," in *Joint Int. Workshop on Interactive Distributed Multimedia Systems/Protocols for Multimedia Systems (IDMS/PROMS)*, p. to appear, Springer-Verlag, Berlin, November 2002.

[3] M. Ruffin, "An survey of logging uses," tech. rep., Dept of Computer Science, University of Glasgow, Glasgow, Scottland, February 1995. http://citeseer.nj.nec.com/ 36315.html.

[4] W. Shapiro and R. Vingralek, "How to manage persistent state in DRM systems," in *Proceedings of the ACM Workshop in Security and Privacy in Digital Rights Management*, pp. 176–191, November 2001. http://www.star-lab.com/sander/spdrm/papers.html.

[5] M. Bellare and B. S. Yee, "Forward integrity for secure audit logs," tech. rep., UC at San Diego, Dept. of Computer Science and Engineering, Novem-

ber 1997. http://citeseer.nj.nec.com/ bellare97forward.pdf.

[6] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, ch. 12. CRC Press, 2001. ISBN: 0-8493-8523-7.

[7] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," in *ACM Transactions on Information and System Security*, vol. 2, pp. 159–176, ACM Press, May 1999.

[8] B. Schneier and J. Kelsey, "Automatic event-stream notarization using digital signatures," in *Security Protocols, International Workshop Proceedings*, pp. 155–169, April 1997. http://www.counterpane.com/event_stream.html.

[9] J. Kelsey and B. Schneier, "Minimizing bandwidth for remote access to cryptographically protected audit logs," in *Web proceedings of the 2^{nd} International Workshop on Recent Advances in Intrusion Detection (RAID'99)*, 1999. www.raid-symposium.org/raid99.

[10] T. Sander and C. F. Tschudin, "Protecting mobile agents against malicious hosts," in *Mobile Agents and Security, LNCS 1419*, (Heidelberg, Germany), pp. 44–60, Springer-Verlag, 1998.

[11] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," tech. rep., Department of Computer Science. The University of Auckland, July 1997. http://www.cs.arizona.edu/ collberg/ Research/Publications/ CollbergThomborsonLow97a/.

[12] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in 25^{th} *Principles of programming languages (POPL)*, (San Diego, California, United States), pp. 184–196, ACM Press, New York, January 1998.

[13] M. Jakobsson and D. M'Raihi, "Mix-based electronic payments," in *Selected Areas in Cryptography*, pp. 157–173, Springer-Verlag, Berlin, 1998.

[14] B. Schnoenmakers, "Basic security of the ecash^{TM} payment system," in *State of the Art in Applied Cryptography, Courses on Computer Secrutiy and Industrial Cryptography, Leuven, Belgium*, p. 16, Springer-Verlag, Berlin, June 1997.

[15] W. Aiello, A. D. Rubin, and M. Strauss, "Using smartcards to secure a personalized gambling device," in *ACM Conference on Computer and Communications Security*, pp. 128–137, ACM Press, 1999.

[16] X. Leroy, "On-card bytecode verification for java card," *Lecture Notes in Computer Science*, vol. 2140, pp. 150–164, 2001.

[17] D. Clausen, D. Puryear, and A. Rodriguez, "Secure voting using disconnected, distributed polling devices," tech. rep., Standford University, Dept. of Computer Science, June 2000. http://dave.47jane.com/voting/cs444n_voting_report.pdf.

[18] J. Dyer, R. Perez, S. Smith, and M. Lindemann, "Application support architecture for a high-performance, programmable secure coprocessor," in *22^{nd} National Information Systems Security Conference*, October 1999. http://citeseer.nj.nec.com/dyer99application.html.

[19] P. Gutmann, "An open-source cryptographic coprocessor," in *Proceedings of the 9^{th} USENIX Security Symposium*, August 2000. http://www.cryptoapps.com/ peter/usenix00.pdf.

[20] R. Weis and S. Lucks, "The performance of modern block ciphers in java," in *Proceedings of Third International Conference, CARDIS'98*, pp. 125–133, Springer-Verlag, Berlin, September 1998.

[21] B. Schneier and D. Whiting, "Twofish on smart cards," in *Proceedings of Third International Conference, CARDIS 98*, pp. 265–276, Springer-Verlag, Berlin, September 1998.

[22] A. Durand, "Efficient ways to implement elliptic curve exponentiation on a smart card,"

in *Proceedings of Third International Conference, CARDIS 98*, pp. 357–365, Springer-Verlag, Berlin, September 1998.

[23] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, vol. 3, no. 2, pp. 99–111, 1991.

[24] J. B. Knudsen, *Java Cryptography*. O'Reilly United Kingdom, 1998. ISBN: 1565924029.

[25] F. P. 180-1, "Secure hash standard," tech. rep., US Departmenet of Commerce/NIST, Washington D. C., United Stats, April 1995.