License Protection with a Tamper-Resistant Token*

Cheun Ngen Chong[1], Bin Ren[1], Jeroen Doumen[1], Sandro Etalle[1,2], Pieter H Hartel[1], and Ricardo Corin[1]

[1] EEMCS Faculty, University of Twente, Postbus 217, 7500 AE Enschede The Netherlands
{chong,ren,doumen,etalle,pieter,corin}@cs.utwente.nl
[2] Center for Mathematics and Computer Science (CWI), P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Sandro.Etalle@cwi.nl

Abstract. Content protection mechanisms are intended to enforce the usage rights on the content. These usage rights are carried by a *license*. Sometimes, a license even carries the key that is used to unlock the protected content. Unfortunately, license protection is difficult, yet it is important for digital rights management (DRM). Not many license protection schemes are available, and most if not all are proprietary. In this paper, we present a license protection scheme, which exploits tamper-resistant cryptographic hardware. The confidentiality and integrity of the license or parts thereof can be assured with our protection scheme. In addition, the keys to unlock the protected content are always protected and stored securely as part of the license. We verifiy secrecy and authentication aspects of one of our protocols. We implement the scheme in a prototype to assess the performance.

1 Introduction

In a digital rights management (DRM) system, we use a *license* to specify the rights of a user on digital content [8]. For example, a commercial software license restricts the execution of the licensed software to a particular number of times. A license often carries the key to unlock the protected content. Additionally, the license also carries *metadata* of the content, which may be as valuable as the content itself. Therefore, the license, just as the content, requires adequate protection.

Unfortunately, license protection does not attract as much attention as content protection. There are only a few license protection schemes available, and most if not all are proprietary.

Our main security objectives are to ensure confidentiality and integrity of a license or parts thereof, so that keys and metadata can be protected. Additionally, we would like to enforce different usage rights on different parts of the content. For instance, a patient record contains sensitive information about a patient. We want to protect and share this information by using different keys. The doctor is issued all keys to access the entire patient record, but the insurance agent is only issued some keys to access some necessary information. To protect the patient record from being misused, we need to protect these keys.

* This project is funded by Telematica Instituut, The Netherlands.

In this paper, we propose a license protection scheme using a key tree and a tamper-resistant hardware token with which we are able to achieve the aforementioned objectives. A hardware token provides a tamper-resistant environment for storage and cryptographic operations; while a key tree grants us the flexibility to protect and share a license and content.

Our contributions can be listed as follows:

1. We propose a license protection scheme using a key tree and a hardware token, which is able to protect the license or parts thereof and content parts.
2. We perform an analysis of our protection scheme to justify its security properties.
3. We implement and evaluate the license protection scheme by using some off-the-shelf software tools and a Java iButton.

We have applied our license protection scheme in a number of usage scenarios. To explain our approach, we choose a scenario of stock price, where a provider (e.g. NYSE) issues a license that restricts access of brokers (i.e., paid subscribers) and normal users to specific information on stock prices.

The organization of the remainder of the paper is as follows: Section 2 lists the security requirements. Section 3 briefly explains LicenseScript. Section 4 discusses our license protection scheme. Section 5 explains our prototype implementation. Section 6 reports on a performance evaluation of the prototype to justify the applicability. Section 7 briefly explains some related work. Finally, section 8 concludes and suggests future work.

2 Security Requirements

We assume that some of the system components can be trusted. This is more or less realistic with consumer electronic (CE) devices, but much less realistic when working on personal computers. In particular, we assume that the application interprets a license correctly. We treat this trusted part of the application as a *reference monitor* [14]. For example, as soon as the license expires, the application stops rendering. However, a malicious application can still cheat by tampering with the license. Therefore, we define the following requirements for our license protection scheme:

Requirement 1 License Integrity: The application must verify the integrity of the license when it accesses the license.

Requirement 2 Token Interaction: The application must interact with the hardware token to access the license and content parts.

Requirement 3 Keys Confidentiality: The storage keys for accessing the license and content parts must be hidden from the application.

When these requirements are fulfilled, cheating by tampering with the license will be made difficult.

3 LicenseScript License

In this section, we discuss our licensing language, LicenseScript. The language is based on multiset rewriting [3] and logic programming [12]. The reader may refer to our previous work [5] for more detailed information.

A license has the following form:

```
license(Content,Clauses,Bindings)
```

Here, `Content` is (a link to) the content to be protected; `Clauses` is a Prolog program that decides if the operations performed are allowed or forbidden; and `Bindings` is a list of attributes that carry the status of the license and metadata of the content.

A clause has the following form:

```
Head :- Body_1,Body_2,...,Body_n.
```

Here, `Head` is the name and arguments of the clause, and the conjunction of `Body_1` up to `Body_n` is the body of the clause.

We use the scenario of stock price (as discussed in section 1) for illustration. Figure 1 is an example of LicenseScript license that allows a broker to view a stock price for 10 times. The license also allows the provider to reset the number of times the stock price is viewed. Finally, the provider can update the stock price.

In Figure 1, `stock_price` is the link to the stock price; `get_value(X,Y,Z)` gets the value of the binding `Y` from the binding list `X` and unifies it with the variable `Z`; `set_value(V,X,Y,Z)` sets the value of the binding `X` from the binding list `V` with the value `Y` and stores the binding into the binding list `Z`; `is_member(X,Y)` checks if element `X` is a member of set `Y`; `get_curr_time(X)` gets the current time and stores it in `X`; `viewed` is the binding that stores the number of times the stock price has been viewed; `maxviews` stores the maximum number of times the stock price can be viewed; `updated` records the time that `stock_price` is updated; `subjects` is the access control list of subjects that can view `stock_price`.

In all clauses, `S` represents the subject making the query, `B1` is the current set of bindings and `B2` is the set of bindings resulting from a successful query. A failed query does not update the bindings. Clauses are triggered via external actions, for example if the `broker` presses the view button on the user interface, the `canview` clause is activated, with the appropriate settings for `S` (i.e., `broker`) and the bindings `B1` and `B2`.

4 License Protection Scheme

In this section, we introduce our license protection scheme. We use the architecture shown in Figure 3.

Four components are involved: the *application*, *reference monitor*, *token*, and *provider*. The application is a piece of software that interacts with the token, and which is used to access the license and the associated content. The reference monitor, which is a *trusted* part of the application, coordinates the actions of the application and the license.

```
01)license(stock_price,
02)[(canreset(S,B1,B2):-
03)   S==provider,
04)   set_value(B1,viewed,0,B2)),
05) (canupdate(S,B1,B2) :-
06)   S==provider,
07)   get_curr_time(T),
08)   set_value(B1,updated,T,B2)),
09) (canview(S,B1,B2) :-
10)   get_value(B1,subjects,Ss),
11)   is_member(S,Ss),
12)   get_value(B1,viewed,X),
13)   get_value(B1,maxviews,Y),
14)   X <= Y, X = X + 1,
15)   set_value(B1,viewed,X,B2))],
16)[maxviews=10,
17) viewed=0,
18) updated=01012004,
19) subjects=[broker] )
```

Fig. 1. A license that restricts a broker to access a stock price under 10 times.

```
license(stock_price,
[(canreset(S,B1,B2):-
   cipher("CJ...", skey1)),
 (canupdate(S,B1,B2) :-
   cipher("XY...", skey3)) ],
 (canview(S,B1,B2) :-
   cipher("AB...", skey4)),
[maxviews=cipher("12...",skey4),
 viewed=cipher("AC...",skey4),
 updated=01012004,
 skey1=cipher("89...",rootkey),
 skey2=cipher("aC...",rootkey),
 skey3=cipher("CC...",skey1),
 skey4=cipher("KL...",skey2),
 mac=cipher("XA...",rootkey),
 subjects=[(provider,rootkey),
           (broker,skey2),
           (alice,skey4)]]
```

Fig. 2. Protected license of Figure 1, storing the storage keys.

Two protocols support the communication between the components. Protocol A is used to send a protected license to the application from the provider. The provider generates the protected license and depending on its business model, decides which part of the license needs to be protected.

Protocol B is used when the application starts using the content, and when the reference monitor interprets the protected license. We will elaborate these protocols later in section 4.3. To use the license, the application must interact with the token and the reference monitor.

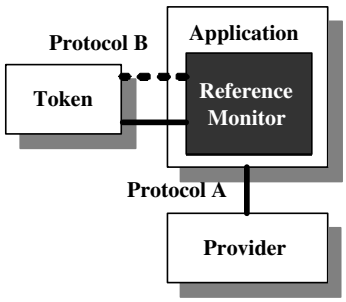

Fig. 3. Overall license protection architecture.

We will explain our scheme as follows: Firstly, we look at protected storage mechanisms, which have inspired our license protection scheme in section 4.1. Secondly, we show the structure of the protected license in section 4.2. Lastly, we illustrate the protection scheme protocols that we have developed in section 4.3.

4.1 Protected Storage Mechanisms

Our license protection scheme is a *protected storage mechanism*. Protected storage is defined by Pearson et al [13] as follows:

> Protected storage is a service to the host platform in which the trusted platform module (TPM) acts as a portal to confidential data stored on an arbitrary, unprotected storage media.

Here, the TPM is a tamper-resistant cryptographic hardware module that is permanently embedded in a computer. The TPM can provide secure storage for keys and other sensitive information and it can perform cryptographic operations. Our license protection scheme uses an external hardware token instead of the TPM because this allows user the freedom to move licenses and content between machines.

In this paper, we use protected storage in the form of a *key tree*. This is a mechanism that has been used in secure group communication for key distribution and management [11].

Figure 4 provides an example of a key tree. A child node is encrypted using the storage key of the parent node. The root key is the "master key" for the whole tree. If say, `skey1` is needed to decrypt `data1`, the former will be decrypted using the `rootkey`. Then, `data1` can be decrypted with `skey1`.

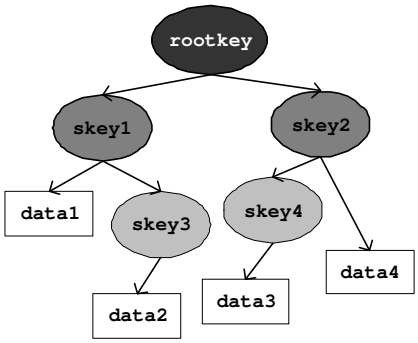

Fig. 4. An example of key tree.

For optimal performance, we use symmetric keys for the root key and the storage keys. The root key is stored on the token when it is issued and never leaves the token. It is sent to the user physically with the token. This root key is the secret key shared between the token and the provider. All decryptions take place on the hardware token for maximum security.

However, when sharing license information with another user, an actual storage key (which has become the root key for a sub-tree) must be transferred to the user's token. For instance, we can allow a user to *only* access `data3` and `data4` by transferring the actual storage key `skey2` to the user's token. The process of transferring this storage key to another user's token falls outside the scope of this paper. However, we believe it can be achieved with simple and secure transfer mechanisms, such as mechanisms of Aura and Gollmann [2] or Atallah and Li [1].

We can selectively deploy the information of the license with other entities by using a key tree. For instance, we can share the information of the license encrypted by `skey3` and hide the others from another user, by using the key `skey2` as the root key for that user.

4.2 Protected License

By using a key tree, we can protect the license from Figure 1, as shown in Figure 2. Here, `cipher(X,Y)` is a predicate that stores the encrypted value `X` (which looks meaningless to human eyes) with the key `Y`.

Several additional bindings are needed to store the encrypted keys. For example, `skey1` is the encrypted key used to encrypt lines `03–04` of Figure 1, shown as `"CJ..."` in line `03` of Figure 2. The length of `"CJ..."` is the same as the original text shown in lines `03–04`. We use the binding `mac` (line `15`) to store the message authentication code, i.e., `"XA..."` of the license, which can be verified by using the `rootkey`. The `mac` can ensure the integrity of the entire license.

The `provider` has the `rootkey` (on her token). In other words, the `provider` can access all the encrypted data in the license. Therefore, the `provider` can execute clause `canreset`, which resets the value of the binding `viewed`; clause `canview`, which views the content of `stock_price`; and clause `canupdate`, which updates the content of `stock_price`. On the other hand, the `broker` can only execute the clause `canview`, because her token only has the actual key `skey2`.

In addition, we use the keys in this license to protect some information (i.e., parts) of the `stock_price`. The license only allows an authorized user (with the correct key) to access these protected parts. In Figure 2, `broker`, who is a paid subscriber, can access the `stock_price` information that is encrypted with key `skey2` and `skey4`. The user `alice`, who is not a paid subscriber can access less information, which is encrypted with key `skey4`, of the `stock_price`.

As another example of the flexibility of the scheme, imagine a doctor issuing the right storage key to an insurance agent. The agent can access the necessary information in a patient record, which is encrypted under this storage key, while other sensitive information remains hidden from the agent.

4.3 Protocols

In this section, we describe the protocols of our protection scheme: Protocol A (for transmitting the license) and Protocol B (for using the license). For the reader's convenience, we list the notation we use to describe the protocols in Table 1.

Protocol A This protocol (Figure 5) requires interaction between the hardware token, the application, and the license provider. Its two main objectives are : (1) to send the protected license to the application; and (2) to send the public key of the application to the token. The application's public key will certify the trustworthiness of the application when the license is used.

A1 Application (A) asks Token (T) to get the desired license (identified by "$name$") from Provider (P). T must recognize P.

A2 T generates a fresh nonce N, encrypts N with the shared secret key with P, $K_{(P,T)}$, concatenates N, $\{N\}_{K_{(P,T)}}$ and identity T to the message received from A, encrypts the result with Provider's public key K^+_{eP}, and send this to A. The fresh nonce is necessary to prevent a replay attack. The secrecy of the nonce can be assured by encrypting it with K^+_{eP}. The authenticity of the nonce (i.e., that it was produced by T) is guaranteed by the presence of $\{N\}_{K_{(P,T)}}$. Therefore, a malicious application cannot fabricate the message of step A2 without the token.

A3 A sends its identity and the message received from T to P.

Symbol	Meaning
A	Application
R	Reference Monitor (the trusted part of A)
T	Hardware token
P	Provider
D	Data, i.e., license clause/binding or content
$\{X, Y\}$	Concatenation of X and Y
$\{\cdots\}_K$	Message is encrypted by key K
K_{st}	Storage key
K_{ss}	Session key
$K_{(X,Y)}$	Shared secret key of X and Y
K^+_{eX}, K^-_{eX}	Public and private key of X for encryption
K^+_{sX}, K^-_{sX}	Public and private key of X for signature
$S(M)_{K^-_{sX}}$	Signature of M with K^-_{sX}
$MAC(M, K)$	MAC of message M with K
Lic	License
Key	List of encrypted storage keys

Table 1. The notation.

A4 If P can decrypt the received message, P is implicitly authenticated by T. P increments N, concatenates N with A's public key K^+_{eA} and encrypts the result with T's public key K^+_{eT}. P sends this message and the protected license Lic to A.

A5 A sends the encrypted message to T but stores Lic. The license can be stored securely because its content is protected by a key tree.

$$\text{A1. } A \rightarrow T : \{A, P, \text{“}name\text{”}\}$$
$$\text{A2. } T \rightarrow A : \{N, \{N\}_{K_{(P,T)}}, A, P, T, \text{“}name\text{”}\}_{K^+_{eP}}$$
$$\text{A3. } A \rightarrow P : \{A, \{N, \{N\}_{K_{(P,T)}}, A, P, T, \text{“}name\text{”}\}_{K^+_{eP}}\}$$
$$\text{A4. } P \rightarrow A : \{Lic, \{N+1, A, K^+_{eA}\}_{K^+_{eT}}\}$$
$$\text{A5. } A \rightarrow T : \{N+1, A, K^+_{eA}\}_{K^+_{eT}}$$

Fig. 5. Protocol A – The hardware token, the application and the license provider interact during the transmission of the protected license and the public key of the application.

Protocol B After Protocol A has finished, the application has the protected license. Each time the license is used, Protocol B (Figure 6) is run.

As shown in Figure 3, the reference monitor is assumed to be the trusted part of the application. We trust the reference monitor in the sense that it will correctly interpret

each license. Thus, we assume that the token has obtained and trusted the public key of the reference monitor initially. The steps of Protocol B are:

B1 Application (A) wants to access the license. It initiates the interaction by sending to Token (T) its identity A, the license Lic and the MAC value of the license (retrieved from Lic), $MAC(Lic, K_{(P,T)})$. If validation fails, T terminates the interaction and records the event.

B2 T verifies the integrity of Lic. If the integrity is violated, T terminates the interaction with A, and A cannot access the content. We use secure audit logging to record this incident [7]. If the integrity is validated, T acknowledges A with a randomly generated session key K_{ss}, encrypted with A's public key K^{+}_{eA}. Implicitly, the application is authenticated if the application can read this message using its private key.

B3 A retrieves a list of encrypted storage keys Key needed for the required data $D_{K_{st}}$, and sends it to T. The parameter "$param$" is used to identify the type of D, i.e., if D is a clause, "$param$" is the name of the clause. This message is encrypted with the session key K_{ss} to ensure the authenticity of the session.

B4 Before sending the decrypted data D, two cases are considered:

B4.1 If D is a license binding, T checks it against the previously stored value to assure that D has not been tampered with. If the check fails, the token terminates the transaction. Otherwise, T performs *application-specific* updates on the binding value stored on the token. For instance, the value of binding `played_times` is incremented. In any case, we log the binding values so that when T and A re-connect to P (Protocol A) say, for a new license or content, T sends P the stored binding values, so that P can check if the user has cheated [8]. If D is used for the first time, T will store it (T can trust the integrity of D at the *first time* because D is always encrypted with a storage key).

B4.2 If D is a license clause or a content part, no checking is done due to the limited resources of the token.

Then, T sends D and its signature $S(D)_{K^{-}_{sT}}$ encrypted with a new session key K_{ss_2} to the reference monitor R. The new session key is encrypted with the public key of R, i.e., K^{+}_{eR}.

B5 R verifies $S(D)_{K^{-}_{sT}}$ before interpreting the data D. Then, R sends D to A, encrypted with the session key K_{ss_1}. The encryption with K_{ss_1} is to ensure the authenticity of the session.

B6 After A has used and updated D (i.e., D' is generated), A sends D' to T for re-encryption.

B7 T replies with the encrypted D', i.e. $\{D'\}_{K_{st}}$. T encrypts it with the session key K_{ss_1} to ensure the authenticity of the session.

B8 A new license Lic' is re-constructed by A. A asks T to regenerate a new MAC value for the updated license Lic'.

B9 T sends the new MAC $MAC(Lic', K_{(P,T)})$ to A to finish the final re-construction of Lic'.

Steps B3 to B6 may be repeated in a session for different types of data (i.e., license clause, binding or content part) during the use of the license and content.

This completes the description of the protocols.

B1. $A \rightarrow T : \{A, Lic, MAC(Lic, K_{(P,T)})\}$
B2. $T \rightarrow A : \{K_{ss_1}\}_{K^+_{eA}}$
B3. $A \rightarrow T : \{Key, \{D\}_{K_{st}}, \text{“}param\text{”}\}_{K_{ss_1}}$
B4. $T \rightarrow R : \{\{D, S(D)_{K^-_{sT}}\}_{K_{ss_2}}, \{K_{ss_1}, K_{ss_2}\}_{K^+_{eR}}\}$
B5. $R \rightarrow A : \{D\}_{K_{ss_1}}$
B6. $A \rightarrow T : \{D'\}_{K_{ss_1}}$
B7. $T \rightarrow A : \{\{D'\}_{K_{st}}\}_{K_{ss_1}}$
B8. $A \rightarrow T : \{Lic'\}_{K_{ss_1}}$
B9. $T \rightarrow A : \{MAC(Lic', K_{(P,T)})\}_{K_{ss_1}}$

Fig. 6. Protocol B – The application interacts with the token for using the license.

4.4 Formal Protocol Verification

We have used the protocol verifier CoProVe [9] to verify Protocol A. Basically, what we needed to verify is that a malicious application would not be able to obtain the license without the correct intervention of the token. It is well-known that design of cryptographic protocols is rather error-prone, and that a great deal of published protocols has later been shown to contain errors prejudicing their safety. CoProVe helps at finding possible attacks and at proving that – under certain conditions – a protocol is attack-free. CoProVe works by taking as input a specification of the protocol and a system scenario describing the roles involved in the protocol – in our case token, application and provider of Protocol A – and by analysing all possible interleavings in presence of a malicious intruder.

We adopt two reasonable assumptions to keep our scenario simple yet expressive:

1. The token knows the genuine provider. It is a practical assumption because the token is dispatched by the provider.
2. The provider knows the genuine application. It is also practical because the provider keeps a list of authorized applications that can access the license.

Specifically, we have verified the following properties:

1. Secrecy: A fresh nonce must only be known by the token and the genuine provider. This is to prevent replay attacks.
2. Authentication: The malicious application and provider cannot impersonate the genuine ones. Thereby, the malicious application cannot impersonate a genuine one to decoy the token. We tested that a malicious application could not impersonate the token.

To carry out the verification we had to set up a finite-state scenario (consisting of a finite number of parallel sessions); this is the standard limitation of model-checking approach to verification: while we have checked scenarios with two parallel sessions

(we are going to carry out tests with 3 parallel sessions as well) it is possible – though unlikely – that hidden flaws are revealed only by analyzing scenarios with a higher number of parallel sessions. In the Appendix we report the source code used to check the secrecy of the nonce with two parallel sessions.

4.5 Security Analysis

In this section, we review the requirements of section 2 corresponding to our license protection scheme.

Requirement 1 is satisfied by using a message authentication code. The verification of the MAC value is performed on the token with the root key stored on the token (in Protocol B step B1). Therefore, we can ensure the correctness of the MAC verification because the secret key never leaves the token.

Requirement 2 is fulfilled *if* different parts of the content are encrypted by using different keys stored on the license (in Protocol B step B3). Therefore, the application must interact with the token *continuously* as long as the application accesses the content and license. Our protection scheme is aimed for content is short-lived, i.e., the value of the content is reduced after a short period of time. For instance, stock price. Therefore, once the content has been encrypted and presumably saved in the clear, we do not insist on communication with the token anymore.

Requirement 3 is satisfied. The keys stored on the license are encrypted. The decryption operations (on the keys, license clauses, bindings, and content parts) are performed on the token (in Protocol B step B4). However, during sharing, an actual storage key must be transferred from one token to another. This process is presumed secure by using some available mechanisms.

5 Prototype

In this section, we discuss a prototype implementation of our license protection scheme. The objective is to establish the applicability, and at the same time to conduct some performance evaluation. The prototype is built on a platform of Intel Pentium 4, 256 Mbytes of RAM, and a serial port connection with the iButton version 2.2.

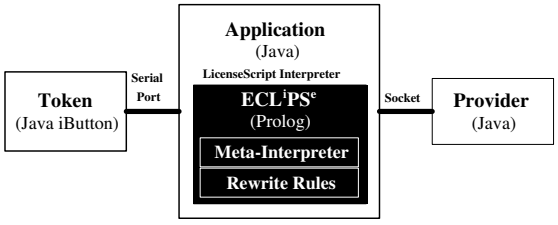

Fig. 7. The architecture and components of the reference implementation.

We use off-the-shelf software tools to implement the components of our prototype, as shown in Figure 7:

1. LicenseScript Interpreter, which is responsible for interpreting and calculating licenses. It *acts* as a reference monitor. We have used the LicenseScript Interpreter from our previous work [6] based on:
   (a) ECL^{i}PS^e: To execute the Prolog code retrieved from the LicenseScript licenses.
   (b) Meta-interpreter: To retrieve the clauses and binding values from the licenses and to send these to the ECL^{i}PS^e Prolog interpreter.
   (c) Rewrite Rules: To interpret the rights operations performed by the users via the application, for instance, play, copy, etc.
2. Application, which is used to access the license and the associated content, while interacting with the token. This is written in Java, using iB-IDE API, Java Cryptography Extension (JCE), and JavaCard Framework.
3. Token, which is a Java iButton version 2.2. It has a higher physical security than a normal smart card because the chip is physically protected by a stainless steel cover, and it supports common cryptographic algorithms.
4. Provider, which provides the protected license and sends it to the client via a socket connection. This is written in Java using JCE and Java.net.

After implementing the prototype, we performed several performance evaluations of the prototype.

6 Performance Evaluation

To verify the practicability of our license protection scheme, we perform several tests on our prototype.

From our previous experience, we know that the cryptographic operations on the iButton are slow [7]. As it is used more frequently than Protocol A, we choose to evaluate the performance of Protocol B, in particular the two operations that involve the iButton: (1) decryption of keys and data (includes license clause, binding and content part) on different level of a key tree, on the iButton (section 6.1), and (2) reconstruction of the license, which involves encrypting the data and generating a message authentication code (section 6.2).

6.1 Test 1: Level of the Key Tree

The depth of a key tree influences the performance of our license protection scheme. For instance, to retrieve a license binding value, which is encrypted with a storage key at level 10 of the key tree, the token has to perform 10 steps of symmetric decryption (including the step to decrypt the encrypted binding value).

In this test, we measure the decryption time required at various levels, i.e., from level 2 to 10 inclusively (level 1 is the root key). The final result obtained for each level is the average of 5 repeated measurements. We run the test as shown in Figure 8.

The size of the data is less than 128 bytes. We found that it takes roughly 0.2 second for DES decryption (with a 64-bit key) on the iButton, which is consistent with the finding of our previous work [7].

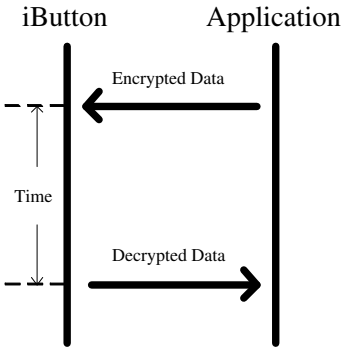

Fig. 8. The procedure for measuring the time needed to perform data decryption at different levels of the key tree.

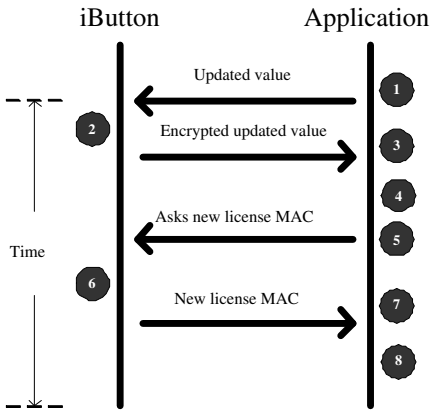

Fig. 9. The procedure for measuring the time needed to perform data re-encryption on the token and reconstruction license on the application, at different levels on the key tree.

We derive a least square fitting (LSQ-Fit) formula to express our result of measurements:

$$t = 0.06 \pm 0.02 + (111.48 \pm 3.08) \times l \quad (1)$$

Here, t is the time in milliseconds required to perform DES decryption on the token for level l on the key tree. $l > 1$ because level 1 is the root key.

The first conclusion is that the depth of the key tree should be kept as low as possible. From Equation 1, it takes approximately 1.22 seconds to decrypt a data (of size less than 128 bytes) at level 10 of the key tree. This will cause a delay to the system, which is noticeable to the user.

6.2 Test 2: License Reconstruction

After the data is used and updated, we also need to re-encrypt the data on the token, reconstruct the license on the application and generate a new MAC for the updated license on the token.

We run a test, as shown in Figure 9. We use the same data size as less than 128 bytes to perform our tests, and we have run the same test for the same data 5 times. The final result is the average value of these 5 tests.

We use an LSQ-fit formula to express our result:

$$t = 2256 \pm 79.96 + (2.56 \pm 0.28) \times l \quad (2)$$

Here, t is the time in milliseconds required to reconstruct the license for updated level l. The time required to reconstruct the license does not depend on the depth of the data in the key tree because only one DES encryption is performed on the iButton. Therefore,

the time required to reconstruct the license for arbitrary updated level in the key tree is approximately 2.25 seconds.

We decompose the procedure of test into 8 parts, as shown in Figure 9, and test the time required for each part:

Parts 1, 3, 5, and 7 Application transmits less than 128 bytes of the data to the iButton and vice versa. We have run a test of the data transfer rate. It takes about 0.2 second to transmit less than 128 bytes of data, as shown in Figure 10. Our data size is about 100 bytes. Therefore, in total the communication between the iButton and the application takes **0.8 second**. Therefore, to update and reconstruct a license,

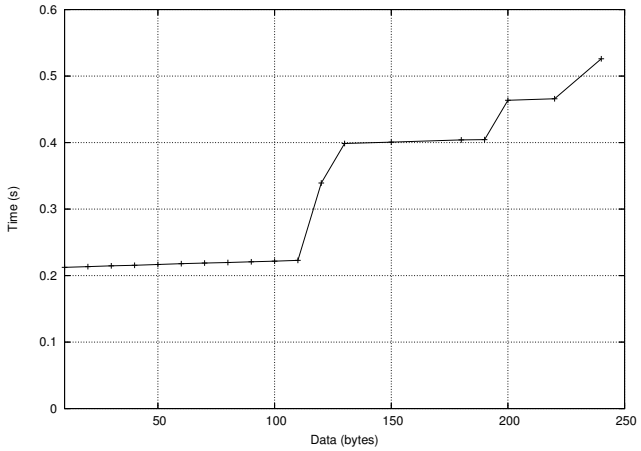

Fig. 10. The data transmission time from the application to iButton.

it takes in total approximately 2.25 seconds, which is consistent with the overall measurement reported at the beginning of this section.

The graph shown in Figure 10 leaps drastically at around 120 bytes of data size. This is due to the iB-IDE API. When the data is over 120 bytes, it will be split into chunks for transfer, which causes more transmit time.

Part 2 This corresponds to Protocol B step B5. The iButton needs to perform a DES decryption with a 64-bit session key on the message to retrieve the updated value, which takes about 0.2 second [7]. Then, the iButton encrypts the updated value with 64-bit storage key, and then with the 64-bit session key. Therefore, this process takes in total **0.6 second**.

Part 4 The application reconstructs the license with the encrypted and updated license data. This takes less than **0.05 second**.

Part 6 This corresponds to Protocol B step B7. Similar to step 2, it takes 0.2 second to decrypt an encrypted message with the session key. The iButton generates a MAC for the new license. The iButton needs about 0.15 second to generate a hash of the data size less than 128 bytes [7]. The iButton needs 0.2 second to generate the

MAC with the root key (DES encryption of the hash). Lastly, the iButton needs 0.2 second to encrypt the MAC with the session key. Therefore, this step takes in total **0.75 second**.

Part 8 The Application reconstructs the license by embedding the new MAC. Similar to 4, the application takes less than **0.05 second** to finish this final step of license reconstruction.

To conclude, the performance of the license protection scheme is acceptable from the user's perspective, if the data is small (less than 128 bytes). We may need a USB interface and a bigger token memory to handle bigger data. This remains for future investigation.

7 Related Work

In this section, we briefly discuss some related work. We investigate some XML documents security that XML-based rights expression languages exploit. We also discuss some commercial license protection mechanisms by using hardware tokens.

Damiani et al [10] define and implement an authorization model for regulating access to XML documents. They exploit the capabilities of XML, and define an XML markup for a set of security elements describing the protection requirements of XML documents. Bertino et al [4] share the objective of Damiani et al but they focus on controlling the data access and dissemination of XML documents when there are XML documents exchanges between two parties. They discuss main protection requirements posed by XML documents and present a set of authorization and dissemination policies to achieve the aforementioned purpose.

As far as we are aware, the listed authorization models only propose the representation of the protected XML documents, e.g. new structure with new set of XML markups, etc. There is no protection operation mentioned how these protected XML documents are produced and accessed.

Several commercial proprietary protection schemes using hardware tokens are available. We are only able to scratch the surface of these mechanisms by studying their white papers. Most of them, e.g. Sospita (`http://www.sospita.com/`) and Wibu (`http://www.wibu.com/`) aim to protect a software code by executing parts of the code on their proprietary hardware tokens. They can lock this part of the software code unless the user pays for it. These protection schemes also assume that (parts of) the application that interfaces with the tokens are trusted.

Basically, protecting a license is only a secondary task in their scheme. Different from LicenseScript, their licenses do not have a rich structure to express complex usage scenarios. In addition, our license protection scheme, because of a key tree, allows flexibility in sharing a protected license and content with other users.

8 Conclusions and Future Work

A *license* is an important element of digital rights management (DRM) because it: (1) specifies a user's rights on a digital content, (2) carries a content key, and (3) describes

metadata of the content. Therefore, we propose a license protection scheme based on a tamper-resistant hardware token and a key tree. The key tree provides flexibility and the hardware token provides tamper-resistance. We apply our license protection scheme to LicenseScript licenses. We analyze the protection scheme in terms of security with respect to some common security assumptions. We also perform a formal protocol verification using CoProVe.

We implement a prototype by using the Java iButton. To justify the practicability, we perform several measurement on the prototype. We conclude that the protection scheme is practical for a shallow key tree and small license size. We will extend our protection scheme for protecting fancy media, e.g. music or film. We will also use a USB connection for the iButton to improve the performance.

References

1. M. J. Atallah and J. Li. Enhanced smart-card based license management. In *IEEE International Conference on E-Commerce*, pages 111–119. IEEE Computer Society, June 2003.
2. T. Aura and D. Gollmann. Software license management with smart cards. In *Proeedings of USENIX Workshop on Smartcard Technology*, pages 75–85. USENIX Association, May 1999.
3. J-P. Banâtre, P. Fradet, and D. L. Métayer. Gamma and the chemical reaction model: Fifteen years after. In C. Calude, G. Paun, G. Rozenberg, and A. Salomaa, editors, *Workshop on Multiset Processing (WMP)*, volume 2235 of *LNCS*, pages 17–44. Springer-Verlag, Berlin, August 2001.
4. E. Bertino, S. Castano, E. Ferrari, and M. Mesili. Controlled access and dissemination of XML documents. In *Proceedings 2nd ACM Workshop on Web Information and Data Management (WIDM'99)*, pages 22–27, 1999.
5. C. N. Chong, R. Corin, S. Etalle, P. H. Hartel, W. Jonker, and Y. W. Law. LicenseScript: A novel digital rights language and its semantics. In K. Ng, C. Busch, and P. Nesi, editors, *3rd International Conference on Web Delivering of Music (WEDELMUSIC)*, pages 122–129, Los Alamitos, California, United States, September 2003. IEEE Computer Society Press.
6. C. N. Chong, S. Etalle, P. H. Hartel, R. Joosten, and G. Kleinhuis. Service brokerage with prolog. Technical Report TR-CTIT-04-14, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, February 2004.
7. C. N. Chong, Z. Peng, and P. H. Hartel. Secure audit logging with tamper-resistant hardware. In D. Gritzalis, S. D. C. di Vimercati, P. Samarati, and S. K. Katsikas, editors, *18th IFIP International Information Security Conference (IFIPSEC)*, volume 250 of *IFIP Conference Proceedings*, pages 73–84. Kluwer Academic Publishers, May 2003.
8. C. N. Chong, R. van Buuren, P. H. Hartel, and G. Kleinhuis. Security attribute based digital rights management (SABDRM). In F. Boavida, E. Monteiro, and J. Orvalho, editors, *Joint Int. Workshop on Interactive Distributed Multimedia Systems/Protocols for Multimedia Systems (IDMS/PROMS)*, volume 2515 of *LNCS*, pages 339–352. Springer-Verlag, November 2002.
9. R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In M. V. Hermenegildo and G. Puebla, editors, *9th International Static Analysis Symposium (SAS)*, volume 2477 of *LNCS*, pages 326–341. Springer-Verlag, September 2002.
10. E. Damiani, S. de C. di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *Advances in Database Technology - EDBT 2000, Proceedings 7th International Conference on Extending Database Technology*, volume 1777 of *Lecture Notes of Computer Science*, pages 121–135. Springer, March 2000.

11. J. Goshi and R. E. Ladner. Algorithms for dynamic multicast key distribution trees. In *Proceedings of of the twenty-second annual symposium on Principles of distributed computing*, pages 243–251. ACM Press, 2003.
12. J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, 1987. Second edition.
13. S. Pearson, B. Balacheff, L. Chen, D. Plaqui, and G. Proudler. *Trusted Computing Platforms – TCPA Technology in Context*. Prentice Hall PTR, Upper Saddle River, New Jersey 07458 United States, 2003.
14. R. Sandhu and J. Park. Towards usage control models: beyond traditional access control. In *7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, pages 57–64. ACM, June 2002.

Appendix: CoProVe for Protocol A

```
% Specification of Protocol A:

% specification for application role
application(A,T,P,N,L,K,[
    send([A,P]),
    recv([N, [N+K, [A, [P,T] ] ] ]*pk(P)),
    send([A, [N, [N+K, [A, [P,T] ] ] ]*pk(P)]),
    recv([L, [sha(N), [A,pk(A)] ]*pk(T)]),
    send([sha(N), [A,pk(A)] ]*pk(T))
]).

% specification of token role
token(A,T,P,N,K,[
    recv([A,P]),
    send([N, [N+K, [A, [P,T] ] ] ]*pk(P)),
    recv([sha(N), [A,pk(A)] ]*pk(T))
]).

% specification of the provider
provider(A,T,P,N,L,K,[
    recv([A, [N, [N+K, [A, [P,T] ] ] ]*pk(P)]),
    send([L, [sha(N), [A,pk(A)] ]*pk(T)])
]).

% secrecy check (singleton role)
secrecy(N, [ recv(N) ] ).

%scenario to check the secrecy of nonce with 2 sessions
scenario([[a1,Init1], %  application
         [a2,Init2], %  application
         [t1,Resp1], % token
         [t2,Resp2], % token
         [p1,Resp3], %  provider
         [p2,Resp4], %  provider
         [sec,Secr1] ]) :-
 application(a,_,p,_,_,_,Init1),
 application(m,_,p,_,_,_,Init2),
 token(_,t,p,n1,k,Resp1), % token knows genuine provider
 token(_,t,p,n2,k,Resp2),
 provider(_,t,p,_,l1,_,Resp3), % provider knows token
 provider(_,t,p,_,l2,_,Resp4),
 secrecy(n1, Secr1).

initial_intruder_knowledge([t,a,m,p]).
has_to_finish([sec]).
```