Protocol and Architecture for the Secure Delivery of High-Value Digital Content

Alexander Truskovsky

A Thesis
in
The Department
of
Computer Science
and
Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

November 2005

© Alexander Truskovsky, 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-14339-8
Our file Notre référence
ISBN: 0-494-14339-8

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

ABSTRACT

Protocol and Architecture for the Secure Delivery of High-Value Digital Content

Alexander Truskovsky

Digital Rights Management (DRM) is used to control access to digitized intellectual property and sometimes to control how that property is used. In the media context, this often involves a player together with a (possibly incorporated) "set-top box". Historically, DRM schemes have been too fragile to protect high-value digital content. In this thesis, we remedy that problem. Through registration, a user's identity is bound to a tamper-proof set-top box storing shared secrets and running a hard-wired program. Encrypted content of interest is obtained by arbitrary means. The user activates the box to engage in a protocol with a remote server operated on behalf of the content owner. The server securely delivers the capability to display this content precisely once and records this fact. Keying information is hidden from the user in such a way that key distribution and authentication are radically simplified, resulting in an extremely robust security architecture.

ACKNOWLEDGMENTS

I would like to thank my supervisor David K. Probst for introducing me to many issues and perspectives in information-systems security, including the sometimes wild and wacky world of digital-rights management (DRM). He provided guidance for this research and was always available to answer my many questions, sometimes at odd hours. Above all, I enjoyed the stimulating discussions we had whenever we met.

I would also like to thank my wife Christine for her endless patience and support while I was working on this thesis. I am aware that the long hours spent on my thesis work took time away from being with my family and I thank my wife for her understanding in this matter.

Table of Contents

List of Figures vii

List of Tables ix

1 Introduction 1

1.1 Classic Cryptography 3

1.2 Unbreakable Cryptosystem (One-Time Pad) 4

1.3 Commercial cryptosystems 4

1.4 Problem of the Thesis 6

1.5 Protocol Security 13

2 Key Establishment Protocols 14

2.1 An Example of a Key Establishment Protocol 14

2.2 Key Establishment Protocol Classification 18

2.3 Use of an on-line server 19

2.3 Use of an off-line server 22

2.5 When the secret is already shared between the participants 25

2.6 Attacks on Key Establishment Protocols 27

3 Rijndael 30

3.1 Input and Output 31

3.2 Encryption 32

3.3 Decryption 34

3.4 The Number of Rounds and Key Schedule 36

3.5 Rijndael Security 38

3.6 Rijndael Experiment 39

4 Tamper-Proof Hardware 41

4.1 Tamper-Proof Box Requirements 42

4.2 FIPS Standard "Security Requirements for Cryptographic Modules" 45

4.3 IBM's secure coprocessor 49

5 Security Architecture for Controlled Access to Digital Content 53

5.1 Security Architecture Overview 53

5.2 Protocol Description 56

5.2.1 Setup 56

5.2.2 Protocol Messages 59

5.2.3 The Protocol in Operation / A State-Transition View 61

5.2.4 Comprehensive Description of the Protocol 63

5.2.4.1 The Client Program 63

5.2.4.2 The Server Program 68

5.3 Protocol Security 73

6 Conclusion 77

6.1 Design Philosophy 78

6.2 Related Work 84

6.3 Future Work 89

References 92

List of Figures

Figure 2.1 Simple Key Establishment Protocol 15

Figure 2.2 Improved Simple Key Establishment Protocol 16

Figure 2.3 Improved Simple Key Establishment Protocol Modified to Withstand Modification Attack 17

Figure 2.4 Needham-Schroeder Shared Key Protocol 19

Figure 2.5 Denning and Sacco Solution 21

Figure 2.6 Beller-Yacobi Protocol 22

Figure 2.7 Potential attack on Beller-Yacobi Protocol proposed by Boyd and Mathuria 24

Figure 2.8 Solution proposed by Boyd and Mathuria 25

Figure 2.9 The Andrew Secure RPC Protocol 25

Figure 3.1 Sample Plaintext Block 31

Figure 3.2 Rijndael Encryption Process 33

Figure 3.3 Round Transformations 34

Figure 3.4 Round Transformations of the Straightforward Decryption Algorithm 35

Figure 3.5 Straightforward Decryption Algorithm for the Two-Round Cariant of Rijndael 35

Figure 3.6 Structure of EqRound and EqFinalRound 36

Figure 3.7 Structure of the Equivalent Decryption Algorithm 36

Figure 3.8 Rijndael Experiment Results 40

Figure 4.1 Tamper-Proof Box 42

Figure 4.4 Hardware Architecture of IBM's High-End Secure Coprocessor 50

Figure 5.1 Protocol Messages 59

Figure 5.2 Protocol States 62

Figure 6.1 Conventional Trust Model (For Communicating).. 77

Figure 6.2 Unconventional Trust Model (For Capability Passing).................................. 78

List of Tables

Table 1.1 Typical Versus Our Approach in the Military Example 10

Table 3.1 N_r as a Function of N_b and N_k 37

Table 3.2 Summary of Attacks on Rijndael 38

Table 4.1 Summary of Security Requirements 47

Table 4.2 Summary of Physical Security Requirements 48

1 Introduction

This thesis proposes a novel security architecture for Digital Rights Management (DRM) of extremely high-value digital content, which maintains its high value for an extended period of time. Like most security systems that are used in e-commerce and elsewhere, our security architecture combines well-known primitives such as cryptographic algorithms and ideas from known security protocols, but combines them in novel ways that is made possible by using an entirely new set of assumptions. In the introduction, we offer a general overview as a lead-up to explaining how our problem and solution is similar to yet differs from previous attempts.

We propose a security architecture, designed in a specific DRM context, although it may have other applicability. The normal DRM context is managing intellectual property by building an architecture for the secure delivery of high value digital content. This involved design of key-exchange protocols, adaptation of cryptographic algorithms, etc...

But what drove us to a different problem from the normal? The two main issues that led us to the new problem are the issues of *scalability* and *trust*.

In a typical case two peer entities that are equally trusted decide to securely communicate a plaintext message. Usually, one is the server that pushes the digital content and one is the client that receives it. However, if there are many clients that want to receive the digital context at the same time, say to view a movie on the opening night, we have a

potential problem of bandwidth because of the volume of the digital content. Thus we were led to consider a low bandwidth solution of "virtual" delivery that preserves scalability.

The basic idea is make the delivery of encrypted content orthogonal to the delivery of the capability to display it. In this way, channels with widely different bandwidths can be used. Once the capability has been delivered over a low-bandwidth channel, the digital content can be displayed exactly once and then the capability vanishes. This gives us a great amount of flexibility in that we are not forced to push many giga bytes of encrypted data through the network.

The second issue is trust. Here our model is changed from a programmable computer plus an untrusted person to a non-programmable, hard-wired, tamper-proof box plus an untrusted person. A number of shared secrets are buried deep inside the tamper-proof box. The end-user has no access to the shared secrets, which allows us to control when and how often the digital content is displayed. This provides us with the ability to securely control the delivery of high value digital context.

The proposed architecture changes the concept of a shared secret. The secrets are known to the server but are not known to the user; they are hidden inside the tamper-proof client box. In short, the secrets are shared between two "boxes", although we do trust the user of the server box. The main consequence of changing the trust model by changing the concept of a shared secret is that secret protection takes on a new flavor. The public key

cryptography is used in our key-transport protocol, but the public keys are not publicly known and are buried inside the hard-wired tamper-proof client boxes. If the integrity of the client box is being compromised, all the secrets buried inside are erased.

We formulated an entirely new DRM problem by making new assumptions that significantly simplify design and implementation. In section 1.4 we give a more detailed example of our DRM problem.

1.1 Classic Cryptography

Since the ancient times, cryptography has been used to protect sensitive information. One of the most rudimentary cryptographic protocols was used by Julius Caesar. The Caesar Cipher is a special case of Shift Cipher, with which messages could be manually encrypted and decrypted. In actual use, cryptographic algorithms are embedded into cryptographic protocols to accomplish specific tasks. Over time, these algorithms evolved and became more complex, and with the invention of computers became computationally intensive. Cryptographic algorithms are usually incorporated into cryptographic protocols that provide solutions for a number of tasks. These protocols, e.g., SSL for secure web browsing, are built from cryptographic algorithms, such as symmetric and asymmetric encryption algorithms, key exchange and establishment algorithms, and message authentication codes (MAC).

1.2 Unbreakable Cryptosystem (One-Time Pad)

The only information-theoretically secure cryptosystem is the One-Time Pad. It was first described by Gilbert Vernam in 1917 and was proved to be "unbreakable" after Shannon developed a concept of perfect secrecy in 1949. Having a One-Time Pad is extremely inconvenient, since it requires having the key length to be at least as long as the length of the plaintext. This complicates the key establishment problem, even assuming we have a genuine random number generator, like a noisy diode. It is the case that government, military or intelligence agencies may prefer the unconditional security the One-Time Pad offers. For example a diplomatic pouch containing the secret keys could be taken to the embassies abroad on regular basis to facilitate secure communication between a finite number of principals. It is still true that the One-Time Pad has no practical commercial application, because of the expense of key distribution. The common thing is symmetric cryptosystems where asymmetric techniques are used to handle key distribution. Indeed, whole new protocols based on asymmetric key cryptosystems have been developed in situations where large number of parties needing to communicate exist, and where different parties have no knowledge of each other prior to communication.

1.3 Commercial cryptosystems

In many commercial, military, and intelligence-community cryptosystems, the principals are not known to each other and the scale precludes usage of "in-person" key establishment protocols. This is so even though cryptosystems that employ in-person key establishment protocols are considered to be generally more secure compared to systems

that do not. Usually, key establishment using standard cryptographic protocols is done dynamically on per session basis, between parties that never meet. Modern sophisticated public-key infrastructures support general key establishment protocols that are scalable and can provide service to a large number, not known a priori, of users. In the jargon, they handle "any-to-any" connectivity. One-Time Pads, on the other hand, were designed to facilitate secure communication between predetermined principals with the ability to distribute the keys out of band.

We will measure the security of a cryptosystem by comparing the cost of breaking it to the value of the information being protected by it. Of course, there is always a possibility that one of the principals may sell the secret key and the data can be decrypted without employing any of the cryptanalytic techniques, which reminds us of the need to pay equal attention to both technical and non technical attacks on our system. Most of the encryption algorithms that are used in commercial cryptosystems are generally computationally secure, in the sense that it is impossible to break the cryptosystem without knowledge of any secrets within a reasonable amount of time. In some cases, when significant analysis has been done, known attacks have been discovered that are much better than exhaustive search through the key space. In other cases, there is not much difference. In such cases, with an ample key space, such as the one provided by Rijndael, this is good security indeed. DES is insecure mostly because its key space is too small. Therefore, if the key space is sufficiently large and known attacks are not significantly better than exhaustive search, the encryption algorithm is considered to be computationally secure. Key establishment protocols, on the other hand, are vulnerable

to various kinds of attacks, e.g., man in the middle attack. Paradoxically, because protocols are usually made more sophisticated to counter a wide range of protocol attacks, they become quite complex and, sometimes, introduce new vulnerabilities that are hard to see because of the complexity.

1.4 Problem of the Thesis

In this thesis, we propose a security architecture that provides secure "virtual delivery" of high value digital content even in a low-bandwidth environment. Our work is in contrast to standard security protocols, which takes for granted a set of assumptions that is reasonable for a pair of trusted peer entities that desire to communicate securely. Because our specific problem differs from problems based on this standard situation we have been led to a quite different set of assumptions, with the result that we can use standard mechanisms in different ways. For example, normally, when RSA is used, we publish n and e, which are the two components of the public key, for every principal. In our case we have been led to explore contexts where publication is not necessary but this does not stop us from using RSA. If we don't publish the values of the public key but keep them secret, then we are not concerned with factorization and other attacks. Since we have a specific problem in which our set of assumptions is appreciably different, this allows us to use classical components in novel ways. The standard assumptions are no longer relevant and the standard attacks are no longer a concern, although the new assumptions might force us to counter new attacks.

The original motivation was the DRM problem mentioned at the very beginning. But much of the solution structure appears general and we would not be surprised if there were several other concrete problems where these or very similar techniques might be used.

We mention that there are a number of important issues that are not addressed in this thesis. First, although we make strong security assumptions about the server, we have not provided a security architecture for the server, including the building in which it sits. Second, although we make equally strong assumptions about the client box, we have not provided a secure manufacturing process.

Third, the security architecture in this thesis should be regarded as a large Internet service; as such, it must deal with concurrency on a large scale, theoretically, up to possibly hundreds of thousands of client boxes that have been registered with a given server, all of whom can request service at the same time.

The server is a parallel computer that – ideally – can execute as many threads as there are registered client boxes. For example, there may be a statically created pool of threads as large as the number of client boxes, and one of these threads is dispatched from the pool when a new protocol instance is instantiated by a request message. If the parallel computer cannot support this many threads – perhaps due to operating-system overhead due to thread scheduling – without performance degradation, the excess concurrency is absorbed by a queue and, although the response time increases with load, the throughput

does not degrade. Another possibility to be considered is to assign distinct server IP addresses to distinct subgroups of client boxes at the time of manufacture. There is considerable freedom here given that the server engages in only very special-purpose computation.

Fourth, this security architecture should be viewed as a "capability service provider". Thus, just as there can be multiple users, so there can be multiple digital-content providers. The capability service provider is merely a middleman between content providers and users, all of whom – in some sense – are its clients.

Our first motivating example is in the area of Digital Rights Management. For instance, there is a new movie opening soon. The movie studio ships copies of the movie to the theaters. On a particular date the movie opens and the theaters are allowed to show it to the public. Often, movies are pirated soon after the opening and illegal copies are sold on the street. Piracy is a problem with the current distribution model. We propose a new distribution model. In parallel to shipping a plaintext movie to theaters, we ship encrypted copies of the movie to individual customers. Alternatively, the customers can download it from studio's web site or pick up a copy at a Blockbuster, etc. Starting on the opening night, the customers can request to play the movie on pay-per-view basis.

We are actually thinking of a problem that would allow the studios to offer an entirely new distribution model. We would like to propose a new technology that would allow the option of having an entirely new distribution system.

This is sort of like conventional pay-per-view. The basic idea is: instead of shipping the digital content, we ship the capability to view the digital content precisely once and then the capability self destructs. The real problem is: we would like to allow somebody in an uncontrolled physical space, but perhaps with a physically controlled box for displaying purposes, to possess the encrypted form of high value digital content in insecure storage. We would then like to invent a protocol, where upon request from the paying customer we would ship a capability to display the content exactly once. This capability can neither be stolen nor copied, and self-destructs at the end of displaying. And if the customer wishes to display the content again, he/she has to go through protocol again - each play is a new request.

Could we find another application? Let's consider protecting software. Our example is from the area of classified programs. We are not sure this example is realistic; however, it does illustrate our point. Assume that there is an agent in the hostile territory. The command station needs to communicate a specific contingency plan to him, depending on the circumstances. For bandwidth reasons, for example to escape detection, the agent would already posses all possible contingency plans in encrypted form. Encrypted contingency plans are safe from everybody including the agent. After being informed which plan is relevant, a simple modification of our protocol has him requesting the permission to view the corresponding contingency plan. He/she would get a capability to decrypt and display a specific plan for a short period of time, and then the capability would vanish. The agent could destroy the box or "zeroize" the secrets in case of

probable capture. Also, the station can deny the request to view any encrypted information once the fact that the agent has been captured has been established. **Table 1** compares how some of the scenarios are handled in the typical approach and our approach.

Typical	Ours
The agent has the knowledge of long-term secrets in plaintext.	The long-term secrets are stored in the tamper-proof box, and the agent has no access to them.
Requires high bandwidth to transmit large messages. Agent may be easier to detect while on-line for too long.	Very small messages, since only keys are sent. Does not require high bandwidth and reduces the chances of being detected.

Table 1.1 Typical Versus Our Approach in the Military Example

Let us summarize the differences more systematically:

1. We are dealing with vast quantities of high value digital content. We want to maintain full control of when and how many times the content is viewed or used.
2. We are not interested in distributing content, which is done some other way, only the capability to display it. The capabilities may go over the Internet, but the protocols use very little bandwidth because relatively few bits need to be exchanged.

3. Trust model: we have a trusted server, and a finite but large number of client boxes, which are considered to be in the hands of untrusted users. We shift the trust from the customer to the client box.
4. The server has a source of true randomness — a true random number generator.
5. Each client box has long-term secrets shared between the server and itself which were uploaded to it at the factory.
6. The box runs precisely one hard-wired program. The client box user has no control over the program's execution. It runs inside the tamper-proof box. Once a request is made, the program always executes from the beginning to the end. If the execution is interrupted, say by power failure, the next request starts the program from the beginning again.
7. We need special tamper-proof hardware that will protect the long-term secrets and the protocol's execution, should someone try to physically steal the secrets. An attempt to tamper with the hardware in order to extract the long term secrets will cause the hardware to destroy all the secret cryptographic information.
8. We need a very secure encryption strategy that is highly resistant to ciphertext only attacks, since the encrypted content will be available without restriction. By very secure encryption we mean that the computer power needed to break the encryption would approach in cost the economic value of the data. We would like to raise the computational cost of breaking the encryption well above of what the thief is going to gain by stealing the movie, so that it becomes not economically interesting to steal the movie in this way.

9. We need a very simple protocol with high assurance that all imaginable attacks on the protocol fail. Key establishment protocols are notorious for slowly discovered bugs. We would like something so simple that we know it is correct from day one, because if the bug is discovered after the system is deployed, it would probably seriously demotivate the studio from continuing.

One major win over the typical security architecture attributes (e.g., the copy-protection schemes that last on the average about a week before they are cracked) is that the exchange of long term secrets is done a priori, at the time the box is manufactured (the client box is subsequently registered with the purchaser). One major problem is that we will have the client boxes in the hands of untrusted users and we have to protect the long-term secrets inside. Here we rely on an extension of standard ideas about tamper-proof hardware. In Chapter 4 we will discus some research in this area and show how it solves our problem of manufacturing a tamper-proof client box.

This thesis tells a simple story which can be summarized in four points:

- We adopt a nuclear threat model in the sense that if even one adversary can defeat out DRM technology, all is lost.
- We adopt a modified trust model in which a registered but untrusted user is bound to a tamper-proof set-top box with shared secrets and a hard-wired program.
- We adopt a capability-based protocol solution in which a client box acquires a display-once capability for user-selected digital content via a protocol with a trusted server.

- We adopt a tamper-proof vault as our architecture solution, in which keying secrets are protected by active countermeasures.

1.5 Protocol Security

What are the consequences of the new problem? The good thing is that the shared secrets are protected inside the tamper-proof client box. Since there is no disclosure of shared secrets, like public keys, many attacks on cryptographic parts of the key establishment protocol will not work. The encrypted messages that will be sent to client will contain the capabilities to display the digital content. The capabilities will consist of the content keys needed to decrypt the digital content. Since the content keys are genuinely random, we are safe from ciphertext only attacks that are based on the statistical regularity and redundancy of a natural language. In particular, the standard statistical cryptanalytic techniques have no force.

What we are doing here is changing a number of trust assumptions that constitute the trust model of the security protocol. We are sharing the secrets not with the client box user but with the client box itself, over which we assume we have total physical control in principle due to the tamper-proof hardware. That is to say we assume that we can "zeroize" the secrets before anyone can steal them. This of course means that attacks that would be possible if the shared secrets were disclosed are impossible. Another radically difference is that we have secrets shared among server and client boxes that are not shared between the human users (the clients).

2 Key Establishment Protocols

A key establishment protocol is an integral part of most security architectures. In this chapter we discuss existing key establishment protocols. Different protocols are used in different environments. We present some existing protocols, classifying them by the environments they are used in rather than using the typical classification criteria.

When two users wish to securely communicate through the use of cryptographic protocols they require a *session key*. A session key is usually used for one instance of the communication protocol and then discarded. After establishing a session key, each party must be certain that the new key is known only to the parties wishing to communicate and perhaps a trusted server, and no one else. The key also has to be fresh.

2.1 An Example of a Key Establishment Protocol

Here is an example that demonstrates a key establishment protocol. We start with a very basic protocol, and then attempt to improve its security, thus demonstrating the challenges the protocol designers have to deal with.

A wishes to communicate with B. A will obtain a fresh session key K_{AB} from server S and transport it to B. This is shown in **Figure 2.1**.

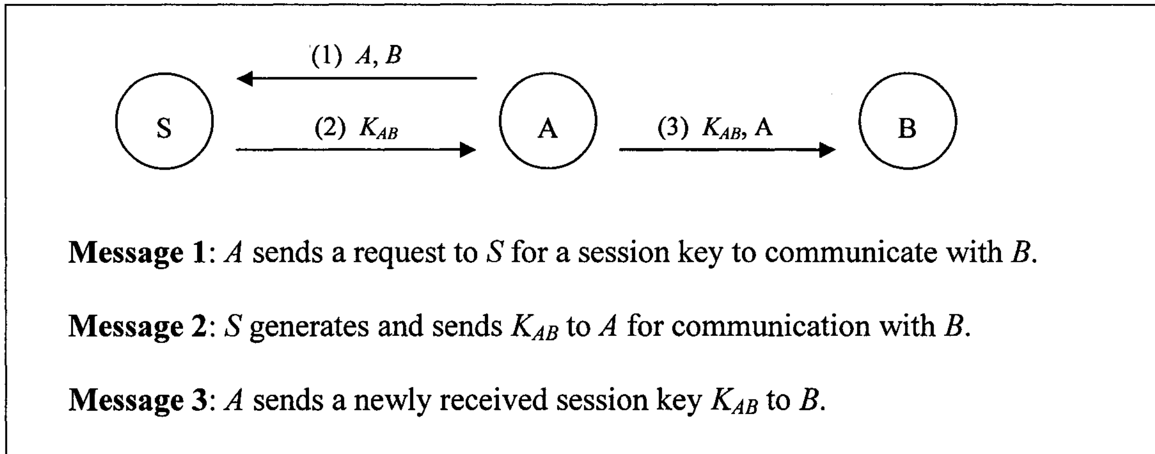

Figure 2.1 Simple Key Establishment Protocol

It is obvious that this protocol is not secure. Anyone may listen in on the conversation and intercept the session key, and subsequently read all the messages A and B exchange. This *Eavesdropping* attack is the most basic of the attacks on key establishment protocol.

Let us now assume that every entity involved in the protocol shares a secret key with the server. With this assumption we try to improve our basic protocol as shown in **Figure 2.2**.

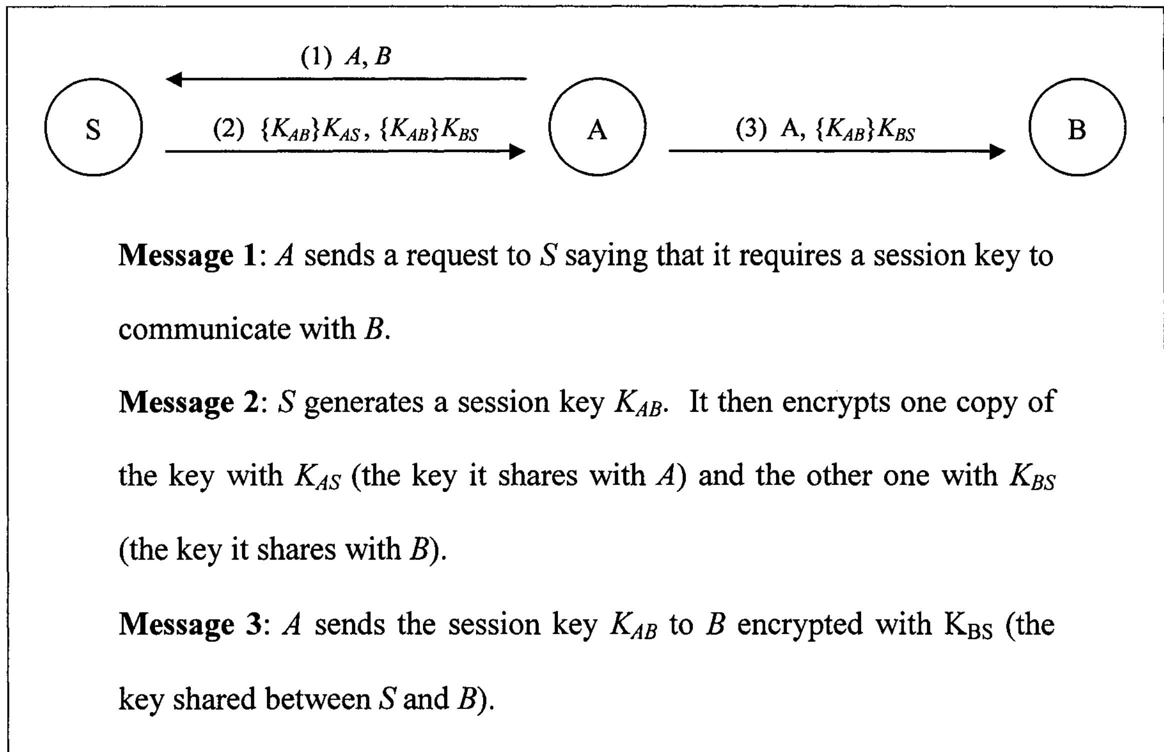

Figure 2.2 Improved Simple Key Establishment Protocol

In this case, we are protected from the eavesdropping attack since each copy of the key is encrypted with the secret key shared between individual parties and the server. However, in the 3rd message, the identity *A* is sent in plaintext and an adversary, say Oscar, can intercept the message and replace *A* with *O*. *B* therefore will believe that it is sharing a session key with *O* and not with *A*. This is an example of a Modification attack. Now, let us modify the protocol to circumvent this attack, as shown in **Figure 2.3**.

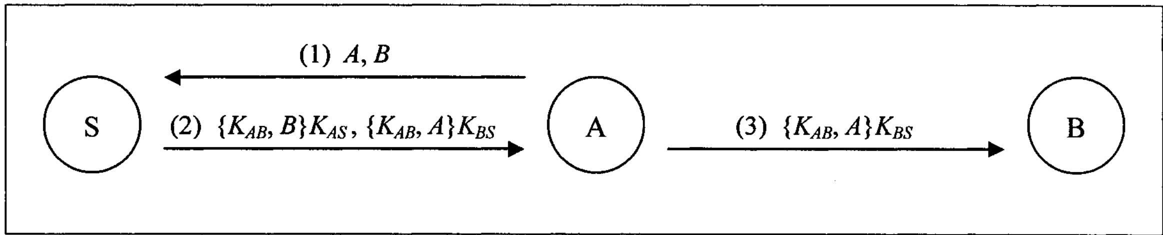

Figure 2.3 Improved Simple Key Establishment Protocol Modified to Withstand Modification Attack

In this version of the protocol, S encrypts K_{AB} along with the identities A and B. Now message 3 cannot be modified since only B (and of course S) can decrypt it.

Here are some goals of key establishment protocols:

- *Entity authentication* is the process of establishing the identity of a second party involved in a protocol, and the aliveness of the second party for a given protocol instance.
- *Data origin authentication* is a type of authentication of a party as the (original) source of data.
- *Key authentication* is where one party is assured that no other party aside from a specifically identified second party may gain access to a particular secret key.
- *Key confirmation* is where one party is assured that a second party actually possesses a particular secret key.

2.2 Key Establishment Protocol Classification

The standard classification is obtained by using two sets of criteria. The first set classifies key establishment protocols as either *key transport* or *key agreement*. In the *key transport* technique one party creates or otherwise obtains a secret value, and securely transfers it to the other. In the *key agreement* technique a secret value is derived from the information contributed by two (or more) parties wishing to communicate. The second set classifies key establishment protocols as either *symmetric* key establishment protocols (i.e., ones that use symmetric encryption) and *asymmetric* key establishment protocols (i.e., ones that use asymmetric encryption).

We would like to classify the key establishment protocols using more practical (i.e., high level) criteria. In order to establish an authenticated session key some existing trust basis must already be available. The principals may already be sharing long term secret keys, or certified public keys may be available. The concept of using public key certificates is considered to be the equivalent of using an *off-line* server. Here are the three possibilities that allow two principals to establish an authenticated session key:

1. The session key is established with the help of an on-line server. In this case each principal shares a long term key with the trusted on-line server.
2. The session key is established with the help of an off-line server. In this case each principal possesses a certified public key.
3. The session key is established with the help of a previously shared secret key between the principals.

We would like to provide one example of a session key establishment protocol for each of the above mentioned categories, along with the typical attacks on these protocols. We will show that the third possibility is generally more secure that the first two. But, first let us describe the types of protocol attacks that exist.

2.3 Use of an on-line server

Server-based key establishment protocols usually use symmetric encryption to establish a session key. All parties that participate in the protocol share a secret key with the trusted server. When one party wishes to communicate with another, they establish a session key through the trusted server, using secrets shared pairwise between individual parties and the server.

As an example, we would like to present the classic protocol proposed by Needham and Schroeder in 1978, which is called the Needham-Schroeder Shared Key Protocol. See **Figure 2.4**.

1. $A \rightarrow S: A, B, N_A$
2. $S \rightarrow A: \{N_A, B, K_{AB}, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$
3. $A \rightarrow B: \{K_{AB}, A\}_{K_{BS}}$
4. $B \rightarrow A: \{N_B\}_{K_{AB}}$
5. $A \rightarrow B: \{N_B - 1\}_{K_{AB}}$

Figure 2.4 Needham-Schroeder Shared Key Protocol

Step 1: A sends to S a message with the following contents:

- The identity A to identify itself
- The identity B to identify a party A wants to establish a session key with
- A nonce N_A to be used as a proof of freshness

Step 2: S sends to A a message encrypted with the private key K_{AS} shared between A and S. The contents of the encrypted message are:

- A nonce N_A as a proof of message freshness
- A session key K_{AB} generated by S
- A message containing K_{AB} and A encrypted with the secret key K_{BS} shared between B and S

Step 3: A sends to B a message encrypted with the private key K_{BS} shared between B and S. The contents of the encrypted message are:

- The session key K_{AB} and the identity A. Only B (and S) can decrypt this message. B is certain that the session key K_{AB} was generated by S and is to be used for communication with A.

Step 4: B sends to A a message encrypted with the session key K_{AB} shared between A and B. The contents of the encrypted message are:

- A nonce N_B to verify that A possesses the same session key

Step 5: A sends to B a message encrypted with the session key K_{AB} shared between A and B. The contents of the encrypted message are:

- The nonce $(N_B - 1)$ encrypted with the session key K_{AB} that proves that A possesses K_{AB}

Before discussing the attack, let us summarize the main idea of the protocol. In message 4, B generates a random number N_B and sends it to A. If A possesses the session key K_{AB} it decrypts N_B, subtracts 1, encrypts $(N_B - 1)$ and sends it back to B. This proves to B that A possesses the new session key K_{AB}.

In 1981 Denning and Sacco pointed out a weakness in this protocol and suggested a solution. This protocol achieves the "good key" property with respect to A (see below). In the second message encrypted with a secret key K_{AS} shared with S, A receives N_A, which is an assurance of key freshness, and identity B, which is an assurance of key authentication. In message 3, B receives A and K_{AB}. However, since there is no proof of message freshness, this protocol is vulnerable to *Replay* attack. If K_{AB} was compromised, the adversary can replay message 3 and make B believe that it shares a session key with A. The proposed solution involves usage of timestamps to prove freshness, as shown in **Figure 2.5.**

$$
\begin{aligned}
&1.\ A \rightarrow S : A, B \\
&2.\ S \rightarrow A : \{B, K_{AB}, T_S, \{A, K_{AB}, T_S\}_{K_{BS}}\}_{K_{AS}} \\
&3.\ A \rightarrow B : \{A, K_{AB}, T_S\}_{K_{BS}}
\end{aligned}
$$

Figure 2.5 Denning and Sacco Solution

2.3 Use of an off-line server

Key establishment protocols that use off-line servers usually employ asymmetric encryption. The off-line server in this case is a certification authority. Each party possesses a signed certificate from the certification authority that proves its identity.

The Beller-Yacobi Protocol, as shown in **Figure 2.6**, is an example of a key transport protocol that uses a certificate to authenticate the end user.

$$
\begin{array}{l}
1.\ A \rightarrow B:\ A, K_A \\
2.\ B \rightarrow A:\ E_A(K_{AB}) \\
3.\ A \rightarrow B:\ \{N_A\}_{K_{AB}} \\
4.\ B \rightarrow A:\ \{B, K_B, Cert(B), Sig(N_A)\}_{K_{AB}}
\end{array}
$$

Figure 2.6 Beller-Yacobi Protocol

Step 1: A sends to B a message with the following contents:

- Identity A to identify itself
- A's public encryption key K_A

Step 2: B sends to A a message encrypted with A's public encryption key K_A. The contents of the encrypted message are:

- A session key K_{AB}.

Step 3: A sends to B a message encrypted with the session key K_{AB}. The contents of the encrypted message are:

- Nonce N_A generated by A to be used as a challenge.

Step 4: B sends to A a message encrypted with the session key K_{AB}. The contents of the encrypted message are:

- Identity of B
- B's public encryption key K_B
- B's certificate
- N_A signed with B's secret signature

After step 4 A is able to ascertain that it established a session key with B. A's challenge (N_A) is signed by B – encrypted with B's private encryption key. A can decrypt it with B's public decryption key and be convinced that N_A was in fact encrypted with B's private key. The owner of B's private key E_B can be verified with *Cert(B)* which is signed by a trusted certificate authority.

This protocol is not completely secure. Here is a potential attack on this protocol proposed by Boyd and Mathuria.

1. $A \rightarrow C_B : A, K_A$
2. $C_B \rightarrow A : E_A(K_{AB})$
3. $A \rightarrow C_B : \{N_A\}_{K_{AB}}$
1'. $C \rightarrow B : C, K_C$
2'. $B \rightarrow C : E_C(K'_{AB})$
3'. $C \rightarrow B : \{N_A\}_{K'_{AB}}$
4'. $B \rightarrow C : \{B, K_B, Cert\ (B), Sig\ (N_A)\}_{K'_{AB}}$
4. $C_B \rightarrow A : \{B, K_B, Cert\ (B), Sig\ (N_A)\}_{K_{AB}}$

Figure 2.7 Potential attack on Beller-Yacobi Protocol proposed by Boyd and Mathuria

In this attack C starts a parallel session with B. In message 4' C obtains B's signature on A's challenge N_A. In message 4 A accepts K_{AB} as a session key between A and B, whereas in fact it is shared with C.

Boyd and Mathuria proposed a simple solution, as shown in **Figure 2.8**, to avoid this attack. Essentially the B should sign the new session key K_{AB} and the challenge N_A in the second message, where N_A guarantees K_{AB}'s freshness. K_{AB}'s confidentiality is protected by a one-way function h. Use of such functions is standard practice in most digital signature schemes. Since K_{AB} is authenticated in the second message, the original fourth message is redundant.

$$
\begin{array}{l}
1.\ A \rightarrow B: A, N_A \\
2.\ B \rightarrow A: E_A(K_{AB}), \{B, K_B, Cert(B)\}_{K_{AB}}, Sig(h(A, B, N_A, K_{AB})) \\
3.\ A \rightarrow B: \{N_A\}_{K_{AB}}
\end{array}
$$

Figure 2.8 Solution proposed by Boyd and Mathuria

2.5 When the secret is already shared between the participants

In this case users wishing to communicate must share individual secrets with each other party participating in the protocol. The session key may either be generated by one party and then transported to the other, or each party may contribute to the generation of the session key.

As an example we present the Andrew Secure RPC Protocol. See **Figure 2.9**. In the first three messages A and B perform a handshake, and in the forth message B sends a new session key to A.

$$
\begin{array}{l}
1.\ A \rightarrow B: \{N_A\}_{K_{AB}} \\
2.\ B \rightarrow A: \{N_A + 1, N_B\}_{K_{AB}} \\
3.\ A \rightarrow B: \{N_B + 1\}_{K_{AB}} \\
4.\ B \rightarrow A: \{K'_{AB}, N'_B\}_{K_{AB}}
\end{array}
$$

Figure 2.9 The Andrew Secure RPC Protocol

Step 1: A sends to B a message encrypted with a secret symmetric key shared between A and B. The contents of the encrypted message are:

- Nonce N_A generated by A

Step 2: B sends to A a message encrypted with a secret symmetric key shared between A and B. The contents of the encrypted message are:

- Incremented nonce ($N_A + 1$) sent by A
- Nonce N_B generated by B

Step 3: A sends to B a message encrypted with a secret symmetric key shared between A and B. The contents of the encrypted message are:

- Incremented nonce ($N_B + 1$) sent by B

Step 4: B sends to A a message encrypted with a secret symmetric key shared between A and B. The contents of the encrypted message are:

- New session key K'_{AB}
- New nonce N'_B

A major flaw was pointed out by Burrows *et al.* Since there is no proof of session key freshness in message *4*, an intruder may substitute a previously recorded message *4* and force A to accept an old (compromised) session key.

2.6 Attacks on Key Establishment Protocols

Since there are infinitely many ways in which an adversary can attack protocols, the following list is obviously not complete. It is however a list of typical (classical) attacks that are known. One must have high assurance, when designing a protocol, that it meets its security objectives given a list of assumptions. It is helpful for the designer to know that a protocol is not vulnerable to the set of known attacks.

The Eavesdropping Attack is considered to be the most basic of attacks. It does not require the adversary to disturb the communications between the legitimate principals, and therefore it is a *passive* attack. It is usually circumvented by using strong encryption.

The Modification Attack involves modification of a message or message fields. Typically cryptographic integrity mechanisms are used to ensure that the message was not tampered with.

The Replay Attack consists of interference with the protocol run caused by inserting a message that had been sent previously in another protocol run.

The Reflection Attack is a special case of a replay attack. This attack requires that parallel runs of the same protocol are allowed. Essentially, in the case of two principals engaged in a shared-key protocol one simply returns a challenge that is intended for itself.

The Denial of Service Attack is an attempt on the part of the adversary to prevent legitimate users from completing the protocol. These attacks are directed at servers that are required to interact with many clients. These attacks are divided into *resource depletion attacks* that use up the computational resources of the server, and *connection depletion attacks* that exhaust the number of allowed connections to this server.

The Typing Attack exploits the fact that, although the protocol elements are clearly distinct when written on a piece of paper, in practice the principal receives a bit string and has to interpret it. For instance, an element that is intended as an identifier is accepted as a key which is a message element of a different type. These attacks typically work with replay of a previous message.

One of the *Cryptanalysis Attacks* is guessing a weak key. Often, the keys are formed from a password that needs to be remembered by a human. Once sufficient evidence is available that suggests how a password has been generated, the attacker can start guessing it and subsequently form a key.

In a *Certificate Manipulation Attack* an adversary gains a certificate asserting that a public key is its own, even though it does not have the corresponding private key.

Finally, a *Protocol Interaction Attack* may be effective if the long-term keys are intended to be used for a single protocol and are in fact used in multiple protocols. For instance, if one protocol is using decryption to prove the possession of an authenticating key, then

the adversary may attempt to use it to decrypt the messages from other protocols. For example, assume server S is using key K to encrypt messages sent to A and also using K to decrypt challenges from B. If B intercepts a message intended for A, it can simply send it to S which will assume the message is a challenge from B and decrypt it.

3 Rijndael

In January 1997, the US National Institute of Standards and Technology (NIST) announced the start of an initiative to develop the Advanced Encryption Standard (AES). The new AES would replace the existing Data Encryption Standard (DES) that was adopted as a standard for "unclassified" applications by the National Bureau of Standards on January 15, 1977. DES is a 16-round Feistel cipher that operates on a 64-bit block with a 56-bit key. After two decades, with the advances in cryptanalysis and faster hardware, there was a need for a new and more secure standard. On October 2, 2000 Rijndael was chosen as the cryptographic algorithm of the AES. AES was adopted as a Federal Information Processing Standard (FIPS) on November 26, 2001 and was to be used only for documents that contain sensitive but not classified information.

The difference between the Rijndael and AES is the range of supported values for the cipher key length and the block size. AES standardized 128-bit block size and 128, 192 and 256-bit key sizes. Rijndael, on the other hand, can accommodate any block size and key sizes that are multiples of 32, as well as changes in the number of rounds that are specified.

3.1 Input and Output

The input and output are one-dimensional arrays of 8-bit bytes. One plaintext block and a key are the input for the encryption, and one ciphertext block is the output. One ciphertext block and a key are the input for decryption, and one plaintext block is the output.

In the encryption phase the plaintext block becomes an intermediate *state* during processing, which at the end of the encryption becomes the ciphertext block. In the similar way, in the decryption phase the ciphertext block becomes an intermediate *state* during processing, and then becomes the plaintext block. The *state* can be pictured as a rectangular array of bytes, with four rows. The number of columns (N_b) is equal to the block length divided by 32. Let the plaintext block be denoted by $p_0p_1p_2 \ldots p_{4*Nb-1}$, where p_0 is the first byte and p_{4*Nb-1} is the last byte. Similarly, a ciphertext block can be denoted by $c_0c_1c_2 \ldots c_{4*Nb-1}$. If we used ASCII to encode the "it is an example" string – one byte per character, then the 128-bit plaintext block would look like this:

I	S		M
T		E	P
	A	X	L
I	N	A	E

Figure 3.1 Sample Plaintext Block

Let the state be denoted by $s_{i,j}$, $0 \leq i < 4$, $0 \leq j < N_b$. The input bytes are mapped onto the state in the following order: $s_{0,0}$, $s_{1,0}$, $s_{2,0}$, $s_{3,0}$, $s_{1,1}$, $s_{2,1}$, $s_{3,1}$, $s_{3,1}$ … . For the encryption, the input is the plaintext and the mapping is $s_{i,j} = p_{i+4j}$, $0 \leq i < 4$, $0 \leq j < N_b$. For the decryption, the input is the ciphertext and the mapping is $s_{i,j} = c_{i+4j}$, $0 \leq i < 4$, $0 \leq j < N_b$. At the end on encryption the ciphertext is extracted from the state: $c_i = s_{i \bmod 4,\, i/4}$, $0 \leq i < 4N_b$. At the end of decryption the plaintext is extracted from state: $p_i = s_{i \bmod 4,\, i/4}$, $0 \leq i < 4N_b$.

Similarly, the key is mapped onto a two-dimensional cipher key. The cipher key can also be pictured as a rectangular array with four rows. The number of columns (N_k) is equal to the key length divided by 32. The bytes of the key are mapped onto the bytes of the cipher key in the following order: $k_{0,0}$, $k_{1,0}$, $k_{2,0}$, $k_{3,0}$, $k_{1,1}$, $k_{2,1}$, $k_{3,1}$, $k_{3,1}$ … . If we denote the key by: m_0, m_1, m_2, … , m_{4*Nk-1}, then the mapping is $k_{i,j} = m_{i+4j}$, $0 \leq i < 4$, $0 \leq j <$ Nk.

3.2 Encryption

Rijndael is a block cipher that performs the repeated application of a round transformation on the state. In **Figure 3.2** we show the encryption process. First the cipher key is expanded into (N_r + 1) sub keys (*expandedKey*) with the KeyExpansion operation. Then the AddRoundKey operation is executed, which takes as input the plaintext and the *expandedKey*. Next is the Round operation, which takes *expandedKey* and *state* as the input and is executed (N_r - 1) times. The last operation on the *state* and

the *expandedKey* is the FinalRound. FinalRound returns the *state*, which is the cipher text.

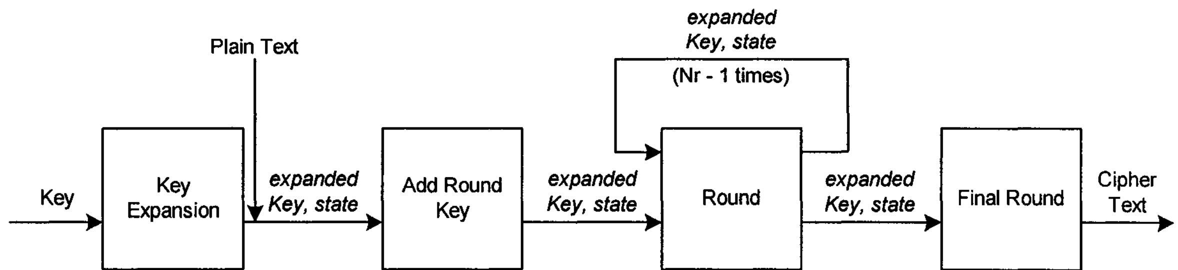

Figure 3.2 Rijndael Encryption Process

KeyExpansion consists of two components: the key expansion and the round key selection. Key expansion specifies how *expandedKey* is derived from the cipher key. AddRoundKey modifies the state by combining it with the bitwise XOR operation and one of the expanded keys. A round key is denoted by *expandedKey*[i], $0 \leq i \leq N_r$. **Figure 3.3** shows the details of the Round and FinalRound transformations. The FinalRound operation differs by one step, namely MixColumns, from the Round operation. The SubBytes is a bricklayer permutation consisting of an S-box applied to the bytes of the state. ShiftRows operation is a byte transposition that cyclically shifts the rows of the state over different offsets. MixColumns is a bricklayer permutation operating on the state column by column.

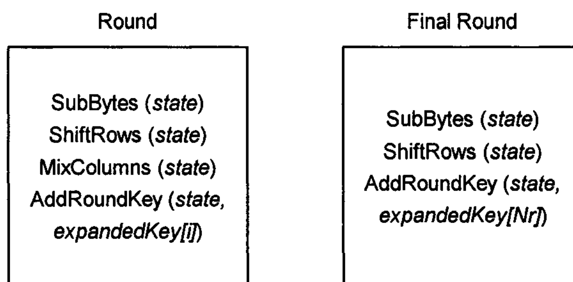

Figure 3.3 Round Transformations

3.3 Decryption

The decryption algorithm can be found in a straightforward way by using the inverse of the steps InvSubBytes, InvShiftRows, InvMixColumns and AddRoundKey in the reverse order. The resulting algorithm is called the *straightforward decryption algorithm.* In this algorithm the steps and their order differ from those used in encryption. For implementation reasons, it is often convenient that the only non-linear step (SubBytes) is the first step of the round transformation. The structure of Rijndael makes it possible to define an *equivalent algorithm for decryption* in which the sequence of steps is equal to that of encryption, with steps replaced by their inverses and a change in the key schedule. Round transformations of the straightforward decryption algorithm are shown in Figure 3.4. Figure 3.5 gives a straightforward decryption algorithm for the two-round variant of Rijndael.

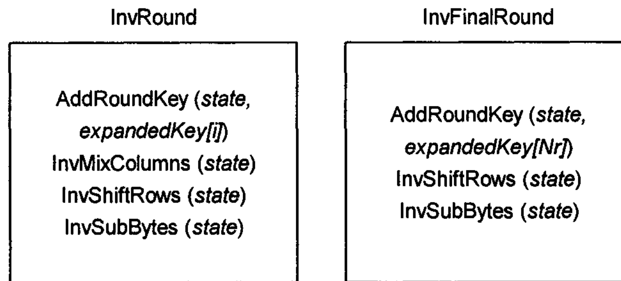

Figure 3.4 Round Transformations of the Straightforward Decryption Algorithm

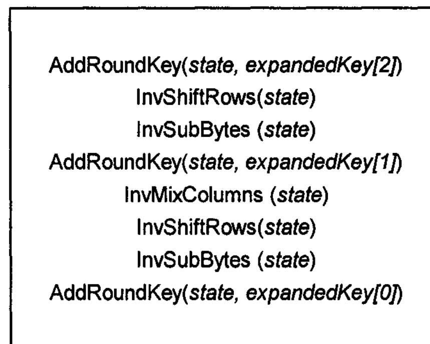

Figure 3.5 Straightforward Decryption Algorithm for the Two-Round Cariant of Rijndael

In order to define the equivalent decryption algorithm the order of InvShiftRows and InvSubBytes is indifferent. InvShiftRows transposes the bytes and has no effect on the byte values. InvSubBytes operates on individual bytes, independent of their position. Therefore, the two steps commute. The order of AddRoundKey and InvMixColumns has to be inverted. It can be done if the round key is adapted accordingly. EqKeyExpansion has been modified to adapt the round key so AddRoundKey and InvMixColumns could to be inverted.

Figure 3.6 shows the structure of EqRound and EqFinalRound. Figure 3.7 shows the structure of the equivalent decryption algorithm.

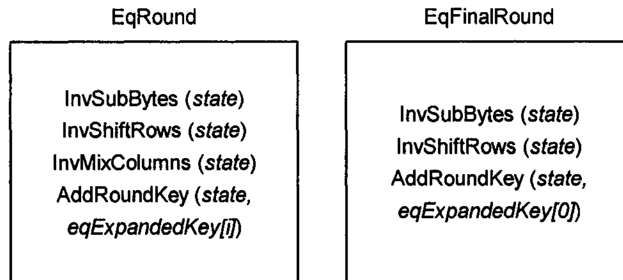

Figure 3.6 Structure of EqRound and EqFinalRound

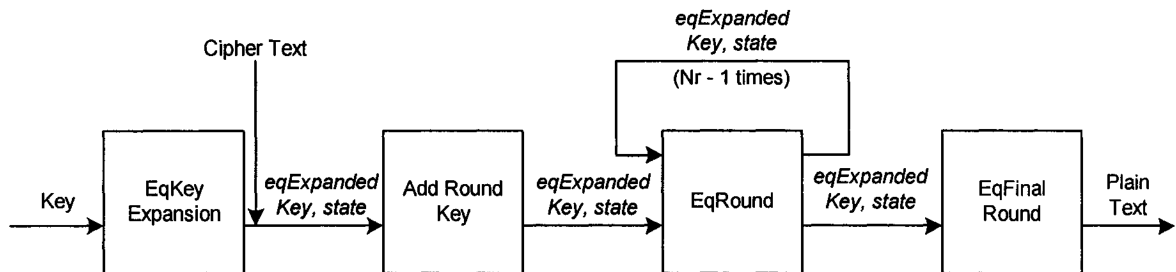

Figure 3.7 Structure of the Equivalent Decryption Algorithm

3.4 The Number of Rounds and Key Schedule

The number of rounds (N_r) is chosen based on the fact that the resistance of iterative block ciphers to cryptanalytic attacks increases with the number of rounds. The designers of the algorithm determined the number of rounds by considering the maximum number of rounds for which shortcut attacks have been found that are significantly more efficient than an exhaustive search and then added a considerable security margin. Table 3.1 gives

the relationship between the block and key size and the number of rounds. If at some point in time the cryptanalytic attacks on iterative block ciphers improve, the number of rounds may be increased in order to increase security at the cost of increasing the computational time.

N_k	N_b				
	4	5	6	7	8
4	10	11	12	13	14
5	11	11	12	13	14
6	12	12	12	13	14
7	13	13	13	13	14
8	14	14	14	14	14

Table 3.1 N_r as a Function of N_b and N_k

(N_b = block length/32 and N_k = key length/32)

The key expansion specifies how *expandedKey* is derived from the cipher key. The number of bits in the *expandedKey* is equal to (block size) * (number of rounds + 1), since the cipher requires one round key for each of the rounds, and one round key for the initial key addition.

3.5 Rijndael Security

Rijndael is secure against all known attacks. There are a number of attacks that have been found against a reduced Rijndael with no more than 9 rounds. The time complexity, although lower that the exhaustive key search, is still too high to have any practical use. A summary of these attacks, including time and data complexities, is described in **Table 3.2**.

Cipher	Key Size	Complexity [Data]	Complexity [Time]	Comments
Rijndael-6	(all)	2^{32} CP	2^{72}	square attack
Rijndael-6	(all)	$6 \cdot 2^{32}$ CP	2^{44}	partial sums
Rijndael-7	(192)	$19 \cdot 2^{32}$ CP	2^{155}	partial sums
Rijndael-7	(256)	$21 \cdot 2^{32}$ CP	2^{172}	partial sums
Rijndael-7	(all)	$2^{128} - 2^{119}$	2^{120}	partial sums
Rijndael-8	(192)	$2^{128} - 2^{119}$	2^{188}	partial sums
Rijndael-8	(256)	$2^{128} - 2^{119}$	2^{204}	partial sums
Rijndael-9	(256)	2^{85} RK - CP	2^{224}	related-key attack

CP – chosen plaintext, RK-CP – related-key chosen plaintext.

Table 3.2 Summary of Attacks on Rijndael [15]

Up to now the versions of Rijndael with a 128-bit block size have been looked at. Rijndael with larger block sizes is different enough that it will have to be analyzed separately. Current techniques have discovered attacks on 7 (of 10) rounds for 128-bit keys, 8 (of 12) rounds for 192-bit key size, and 8 (of 14) rounds for 256-bit keys. Many

of these attacks require virtually the entire codebook of texts and hence are not very practical.

The 9-round related-key attack has a complexity of 2^{224}, which is faster than the exhaustive key search, but is still impractical [15].

3.6 Rijndael Experiment

Since one of the proposed usages of our security architecture involves encrypting images, we wanted to test diffusion in Rijndael. Diffusion refers to rearranging or spreading out the bits in the message so that any redundancy in the plaintext is spread out over the ciphertext. Patterns in an image are easier to spot as compared to text. The goal of the experiment was to see whether there is a partial similarity between the original image and the image produced by decrypting the encrypted image with a key close to the actual key. A good test would be to compare the images decrypted with all the keys in the key range. However, the smallest key size is 32 bit and would generate 2^{32} images. Since we were unable to view 2^{32} images, we generated several hundred images with decryption keys close to the correct key. This experiment does not prove anything, but provides a visual illustration of diffusion in Rijndael. In Figure 3.8 we show an image, originally encrypted with the key = 25, decrypted with key values from 0 through 55. Key = 25 yields the original image and the rest of the keys around 25 yield images that bear no similarity to the original image.

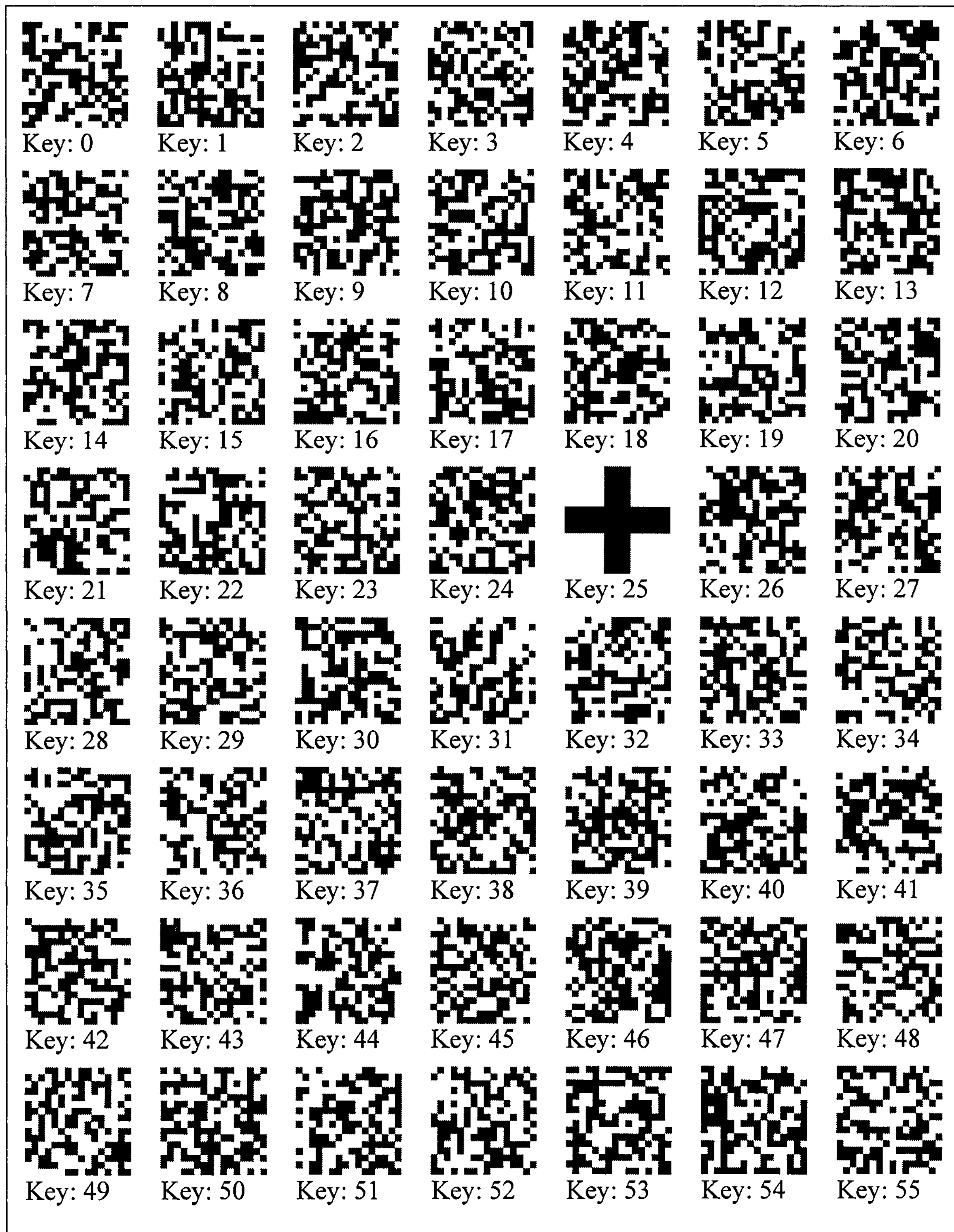

Figure 3.8 Rijndael Experiment Results

4 Tamper-Proof Hardware

The venerable Kirchoff's principle states that you should expect an adversary to learn your encryption algorithm eventually and that your real effort should go into protecting the cryptographic keys. In our security architecture, we have key-like material stored on the server and in physical modules, namely the client boxes, that are not under our physical control. For this reason, one of the requirements of our security architecture is to have the client boxes tamper proof. Detailed design of such tamper-proof boxes is beyond the scope of this thesis. However, in this chapter we would like to present our requirements and solution (4.1), discuss the FIPS standard that manufacturers follow when designing tamper-proof hardware (4.2), and give an example of IBM's secure coprocessor (4.3).

There are several approaches (several levels of security) to physical security designs. One can provide:

- *Tamper evidence*, where packaging forces tampering to leave indelible physical changes.
- *Tamper resistance*, where the device packaging makes tampering difficult.
- *Tamper detection*, where the device actually is aware of tampering.
- *Tamper response*, where the device actively takes countermeasures upon tampering.

For our client boxes we require the highest security level, that is, a robust tamper detection and response mechanism.

4.1 Tamper-Proof Box Requirements

Here we describe the requirements for the tamper-proof box in our security architecture, as well as the chain of our needs and solutions. Our tamper-proof box must provide necessary physical security to protect the key-like secrets inside. Figure 4.1 shows a diagram of the tamper-proof box we have in mind. Upon tamper detection the secrets must be immediately erased (memory must be zeroized). In our protocol, the secrets are divided into short-term secrets and long-term secrets. They are stored differently and therefore have different erasing mechanisms. The protective enclosure must have tamper detection mechanisms built into it. The response to detected intrusion is part of our security design. Upon tamper detection, the box will execute tamper-response measures, namely, it will destroy all the secrets inside. Both tamper detection and secure data deletion are the two most challenging subjects in tamper-proof hardware design.

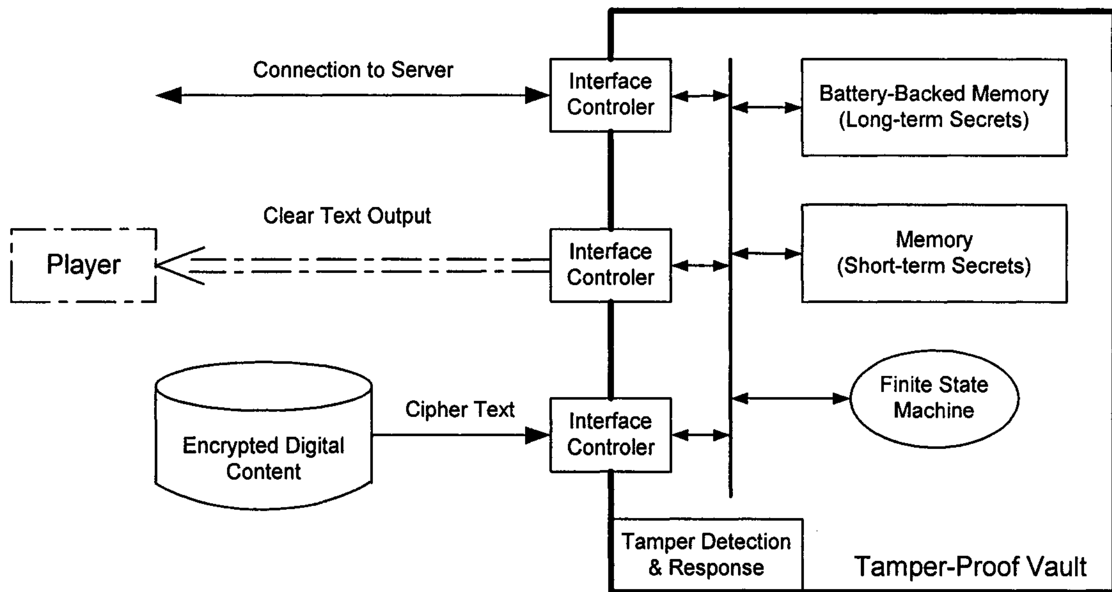

Figure 4.1 Tamper-Proof Box

The short-term secrets are the session keys, the digital content decryption keys, and the plaintext digital content. Short-term secrets are stored in the volatile memory (for example, DRAM) and are erased after protocol termination. Session keys, digital content-decryption keys and plaintext digital content must be protected at all cost. With segmentation, these are also somewhat transient, as material related to earlier segments is overwritten by keys and contents of later segments. We do use multiple content-decryption keys per digital content using segmentation, so it is the case that content-decryption keys are slightly self-protecting; they are overwritten as they are used.

Long-term secrets are stored in the non-volatile memory (for example, battery-backed SRAM) for an extended period of time and are used during some protocol steps. Long-term secrets yield physical bias — they tend to "burn" into memory. We have one technique to overcome bias, but because of non-volatility we require physically active erasing methods for non-volatile memory.

We respond to detected tampering by zeroing out both long-term and short-term secrets. Volatile memory (DRAM), which stores short-term secrets, is zeroized by simply cutting off the power. Non-volatile memory (battery-backed SRAM), which stores long-term secrets, may retain some information because of non-volatility. To make effective zeroization physically feasible, we employ bit-flipping for the long term secrets to reduce bias in the physical medium. Bit-flipping is simply the regular reversal of a bit pattern to be its complement. Assuming bit-flipping, we can effectively zeroize the non-volatile memory as well by cutting off the power, without leaving information residue.

Here are some of the physical principles of bias. Volatile semiconductor memory does not entirely lose its contents when the power is removed. If data is stored transiently, there is no time for it to "burn in". When the same data is stored for long periods of time, it may "burn in". SRAM and DRAM retain some information about the bits that were stored there while the power was still applied. The longer the same information is stored, the more the chip "remembers" what it was after the power-down. In SRAM, the same data stored for extended period of time has the effect of altering the preferred power-up state to the state which was stored when the power was removed. Older SRAM chips could often "remember" the state for several days. If we assume that DRAM can "remember" only if the data was stored for a significant period of time then it will be sufficient to just power it down, but for the SRAM where we know the data will be stored for long periods of time we have to do something more, namely bit-flipping.

Bias is thus dealt with using an active mechanism. We are not troubled by bias for the following simple reason. In order to reduce physical traces of the long-term keys stored in memory for extended period of time (prevent them from burning in), a bit-flipping technique can be used to flip the bits periodically. The goal of this technique is to keep inverting the value stored in order to prevent it from "burning in".

4.2 FIPS Standard "Security Requirements for Cryptographic Modules"

FIPS PUB 140-2 is a publication that specifies the security requirements that must be satisfied by a cryptographic module used within a cryptographic-based security system that provides protection for sensitive or valuable data in Federal organizations. There are four different levels of security intended to cover a range of potential environments. With each increasing level the security requirements are higher. As we see later, we need a variant of *Security Level 4*, the highest level. Although, some of the criteria are not relevant to us in their literal forms, we probably need an analog of each of the things in *Security Level 4*. Two in particular are of special importance. These are tamper detection / response and secure data deletion. We discuss them more in the next section. Table 4.1 summarizes the general requirements while Table 4.2 summarizes the physical security requirements of a cryptographic module. Manufacturers must meet the requirements of a particular security level in order for their device to be used by Federal organizations. There are a number of devices available today from manufacturers such as Cylink, IBM, National Semiconductor, Spyrus, Telequip, and others.

	Security Level 1	*Security Level 2*	*Security Level 3*	*Security Level 4*
Cryptographic Module Specification	Specification of cryptographic module, cryptographic boundary, Approved algorithms, and Approved modes of operation. Description of cryptographic module, including all hardware, software, and firmware components. Statement of module security policy.			
Cryptographic Module Ports and Interfaces	Required and optional interfaces. Specification of all interfaces and of all input and output data paths.		Data ports for unprotected critical security parameters logically or physically separated from other data ports.	
Roles, Services, and Authentication	Logical separation of required and optional roles and services.	Role-based or identity-based operator authentication.	Identity-based operator authentication.	
Finite State Model	Specification of finite state model. Required states and optional states. State transition diagram and specification of state transitions.			
Physical Security	Production grade equipment.	Locks or tamper evidence.	Tamper detection and response for covers and doors.	Tamper detection and response for covers and doors. Environmental failure protection (EFP) or environmental failure testing (EFT)
Operational Environment	Single operator. Executable code. Approved integrity technique.	Referenced Protection Profiles evaluated at Evaluation Assurance Level 2 (EAL2) with specified discretionary access control mechanisms and auditing.	Referenced Protection Profiles plus trusted path evaluated at Evaluation Assurance Level 3 (EAL3) plus security policy modeling.	Referenced Protection Profiles plus trusted path evaluated at Evaluation Assurance Level 4 (EAL4).

<table>
<tr><td rowspan="2">Cryptographic Key Management</td><td colspan="4">Key management mechanisms: random number and key generation, key establishment, key distribution, key entry/output, key storage, and key zeroization.</td></tr>
<tr><td colspan="2">Secret and private keys established using manual methods may be entered or output in plaintext form.</td><td colspan="2">Secret and private keys established using manual methods shall be entered or output encrypted or with split knowledge procedures.</td></tr>
<tr><td>Electromagnetic Interference/ Electromagnetic Compatibility (EMI/EMC)</td><td colspan="2">47 Code of Federal Regulations of Federal Communications Commission (FCC) Part 15. Subpart B, Class A (Business use). Applicable FCC requirements (for radio).</td><td colspan="2">47 Code of Federal Regulations of Federal Communications Commission (FCC) Part 15. Subpart B, Class B (Home use).</td></tr>
<tr><td>Self-Tests</td><td colspan="4">Power-up tests: cryptographic algorithm tests, software/firmware integrity tests, critical functions tests. Conditional tests.</td></tr>
<tr><td>Design Assurance</td><td>Configuration management (CM). Secure installation and generation. Design and policy correspondence. Guidance documents.</td><td>Configuration management (CM) system. Secure distribution. Functional specification.</td><td>High-level language implementation.</td><td>Formal model. Detailed explanations (informal proofs). Preconditions and postconditions.</td></tr>
<tr><td>Mitigation of Other Attacks</td><td colspan="4">Specification of mitigation of attacks for which no testable requirements are currently available.</td></tr>
</table>

Table 4.1 Summary of Security Requirements

	Security Level 1	*Security Level 2*	*Security Level 3*	*Security Level 4*
General Requirements for all Embodiments	Production-grade components (with standard passivation).	Evidence of tampering (e.g., cover, enclosure, or seal).	Automatic zeroization when accessing the maintenance access interface. Tamper response and zeroization circuitry. Protected vents.	Environmental failure protection (EFP) or environmental failure testing (EFT) for temperature and voltage.
Single-Chip Cryptographic Modules	No additional requirements.	Opaque tamper-evident coating on chip or enclosure.	Hard opaque tamper-evident coating on chip or strong removal-resistant and penetration resistant enclosure.	Hard opaque removal-resistant coating on chip.
Multiple-Chip Embedded Cryptographic Modules	If applicable, production-grade enclosure or removable cover.	Opaque tamper-evident encapsulating material or enclosure with tamper-evident seals or pick-resistant locks for doors or removable covers.	Hard opaque potting material encapsulation of multiple chip circuitry embodiment or applicable Multiple-Chip Standalone Security Level 3 requirements.	Tamper detection envelope with tamper response and zeroization circuitry.
Multiple-Chip Embedded Cryptographic Modules	Production-grade enclosure.	Opaque enclosure with tamper-evident seals or pick-resistant locks for doors or removable covers.	Hard opaque potting material encapsulation of multiple chip circuitry embodiment or strong enclosure with removal/penetration attempts causing serious damage.	Tamper detection/ response envelope with tamper response and zeroization circuitry.

Table 4.2 Summary of Physical Security Requirements

4.3 IBM's secure coprocessor

We now give a brief real-world example of a tamper-proof device developed by IBM in 1997. We do so to illustrate how one manufacturer designed a tamper-proof device. Although there are many details in the design, we will mostly concentrate on tamper detection, response and secure data deletion. Figure 4.4 shows the hardware architecture of IBM's high-end secure coprocessor. Two of the design philosophy points that we would like to concentrate on are the following:

- *Battery-backed RAM (BBRAM)* is used as the non-volatile, secure memory
- The device is assembled on a circuit board with technology to actively sense tamper and near instantly *zeroize* the BBRAM

Note that the non battery-backed DRAM corresponds to the volatile memory in which we store short-tem secrets.

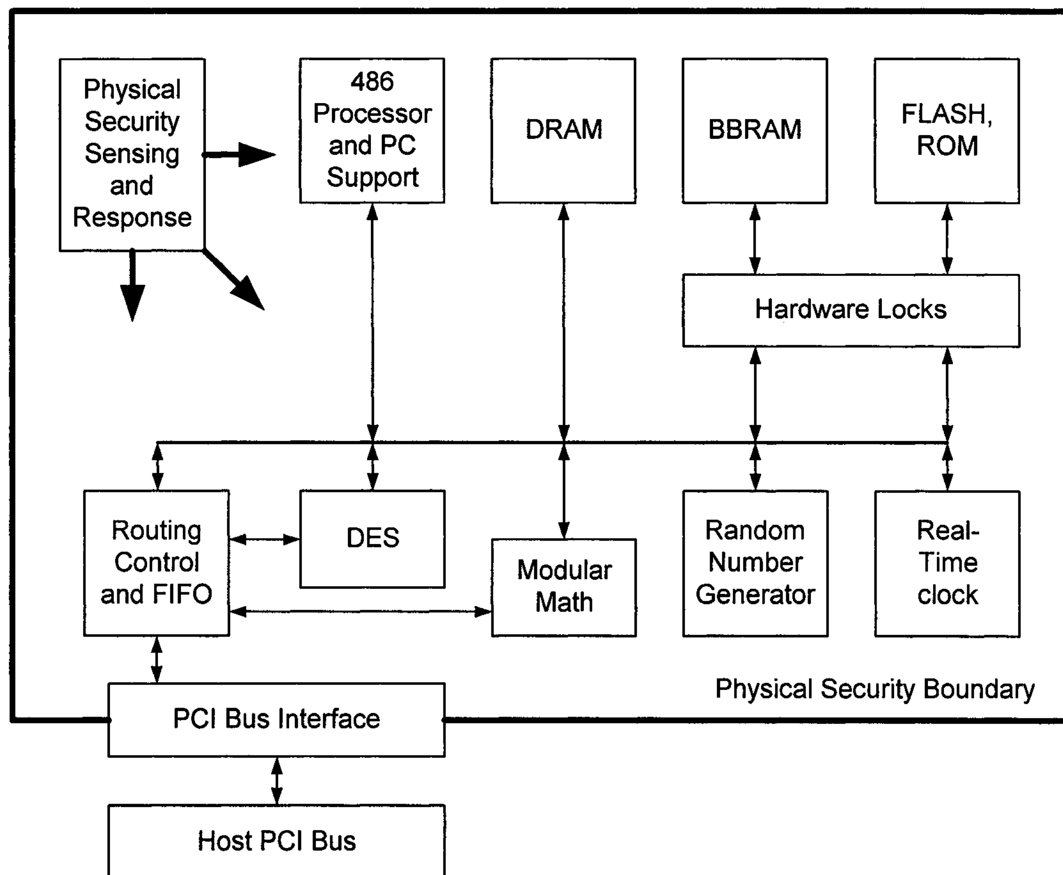

Figure 4.4 Hardware Architecture of IBM's High-End Secure Coprocessor

Tamper detection has a layering approach. The attacker has to go through many layers in order to penetrate the device. The basic element is a grid of conductors that is monitored by special circuitry that can detect changes in the properties of conductors. The conductors are made of the non-metallic material that closely resembles the material in which they are embedded, in order to make discovery, isolation and manipulation close to impossible. The grids are arranged in several layers and the sensing circuitry can detect accidental connection between the layers and changes in individual layer. The grids are made of very flexible material and are wrapped around and attached to the device as if it were being gift-wrapped. After the package is wrapped it is embedded in a potting

material. Since the potting material closely resembles the material of conductors in the sensing grids, it is harder to find the conductors and nearly impossible for an attacker to penetrate the potting without also affecting the conductors. At the end the entire package is enclosed in a grounded shield in order to reduce susceptibility to electro-magnetic interference and to reduce detectable electromagnetic emanations.

Tamper response is to erase (*zeroize*) secrets from the SRAM, erase operating memory and cease operating. SRAM is made persistent with a small battery, and can easily be erased. This battery-backed SRAM (BBRAM) is used as storage for secrets. Upon tamper detection, the BBRAM is zeroized and the rest of the device is held in reset. Tamper detection / response circuitry is active at all times and runs on the same battery as the BBRAM when unit is unpowered. Since tamper can occur quickly, the SRAM must be erased quickly without depending on the CPU being sufficiently operational for sufficiently long to overwrite the contents of the SRAM. To achieve a quick zeroization a simple and effective technique is employed. SRAMs power connection is switched to ground and all data, address and control lines are forced to a high impedance state, in order to prevent back-powering of the SRAM via those lines.

There are some issues that hinder effective zeroization. *Low temperatures* will allow SRAM to retain its data even when the power connection is shorted to ground. To counteract this, a temperature sensor in the device will respond to tamper if the temperature goes below a present level. *Ionization radiation* will cause an SRAM to retain its data, and may disrupt circuit operation. To prevent this, the device detects

significant amounts of ionizing radiation and triggers the tamper response. Storing the same value in a bit in SRAM will cause the value to imprint. For this reason, a bit-flipping software protocol inverts the bits periodically.

5 Security Architecture for Controlled Access to Digital Content

In the previous three chapters we described the concepts and technologies that underlie our security architecture and briefly described some of our assumptions about implementation constraints. In this chapter we make a comprehensive statement about these assumptions and present our contribution – the Security Architecture for Controlled Access to Digital Content, which we describe in detail in this chapter.

5.1 Security Architecture Overview

The goal of our security architecture is the controlled display of high-value digital content. (Only registered players can display content, and one protocol instance is required for each display.) The goal already contains the first assumption, which is that distribution and display can be completely independent. The digital content is encrypted and can be distributed out-of-band. The protocol is not concerned with how the user gets the encrypted digital content. The encrypted content can be downloaded, picked up at a store or ordered by mail, etc.

The digital content is encrypted with a powerful encryption technique and is solid enough so that we can leave it around for a while. Each digital-content entity is encrypted once, with one set of keys. We use a good algorithm, long keys, and enough segmentation (i.e., break the digital file into segments and encrypt each one with the separate key) to make any straightforward cryptanalysis extremely unlikely. Good algorithms, long high-

entropy keys (indeed, truly random ones) and nesting (i.e., iterated encryption) are the three basic sources of strength in any symmetric-key encryption. We will pick these parameters to make it impossibly expensive or impossibly unlikely to crack this encryption. (This is of course relative to the cost of losing the digital content.)

For digital content encryption, we chose Rijndael. This has replaced DES as the new federal standard. It is secure, flexible on the block and key sizes, efficient on a variety of execution platforms. The block and key size of Rijndael can be any multiple of 32. Since Rijndael is a block cipher and we want to use one key on several blocks (in a segment), we chose to use Cipher Block Chaining (CBC), which is a standard choice. Although the plaintext digital content may have some regularity and redundancy, as we saw in Chapter 2, there are no known attacks against Rijndael that are significantly better that the brute-force attack. With a 256-bit key the brute-force attack is not feasible within a reasonable amount of time.

We have chosen to have a rich set of shared long-term key material to allow a very simple (session) key-exchange protocol. These shared long-term keys have to be protected both physically (tamper-proof) and logically (i.e., use of the protocol doesn't expose any shared key to risk). Using this protocol, we distribute the digital content-decryption keys securely so that they can only be used in a "use once" fashion, i.e., these keys are immediately erased upon use.

For initial authentication, we use public-key algorithms without publishing the keys. We have one public-key, private-key pair shared between the server and all client boxes but neither key in the pair is published. (The server has the private key and each client box has a copy of the public key.) All long-term key material is buried either inside one of the tamper-proof hardware client boxes or inside the server. Other key-like material, including a long-term symmetric key, is similarly stored. We chose RSA as our public-key algorithm, which is a standard choice. However, other public-key algorithms may be used, e.g., elliptic-curve cryptography.

The protocol, which may be thought of as a key-transport protocol, involves the request, authentication and the encrypted transport of the content-decryption keys, which is then followed by phase in which the box decrypts the content. The protocol consists of several steps. First, the client box makes a request to display a particular digital-content entity. Second, the server securely distributes the capability (set of crypto keys) to permit display with two design constraints:

1. There is no argument about whether the capability was received.
2. The capability can only be used once and is self-erasing.

By assumption, the server is totally trusted. The client box is trusted if it can confirm its identity using the long-term secrets buried inside it. However, the user of the client box is not trusted. This user has no access to the shared secrets and no access to the hardwired program in the client box that executes the protocol. The box runs the program to completion (or aborts) each time the user initiates it. The program code itself

is not secret but it is hardwired into the box. It executes within a tamper-proof environment and the working memory is secure from the user. Any attempt to gain access inside the tamper-proof space will result in complete erasure of the long-term secrets and the working memory.

As we mentioned in Chapter 1, the goal of this architecture is delivering the capability to play digital content, (where play is to be interpreted abstractly). Some of the examples of the digital content include a film, a game, a program, etc. Once one capability is delivered and the content is decrypted it is displayed as "plaintext". A player must be able to securely play the content without allowing the user to access the "plaintext" bits of the content. The design of such a player and incorporation of the client box into such a player is outside the scope of this thesis. Note that digital content spans many things including entertainment media (films), sensitive programs whose execution needs to be controlled, etc.

5.2 Protocol Description

5.2.1 Setup

Long-term and short-term (mostly keying) material is distributed among the server and the client boxes. Nobody other than the server and the collection of the client boxes has access to this material.

Long-term material (determined at client box manufacture time and known to the server):

- C_{AUTH} – Client's secret identification/authentication string (shared long-term between the server and a particular client box).
- C_{ID} – Client's short ID used to allow lookup optimization (shared long-term between the server and a client box).
- K_S^+ – Server's RSA public key used to encrypt messages sent by the client boxes to the server and known to all the client boxes but otherwise secret. (Shared long-term between the server and all client boxes).
- K_S^- – Server's RSA private key used to decrypt all the incoming messages sent by the client boxes to the server and known only to the server. As always, the keys include the modulus.
- K_{CS} – one Rijndael key shared between the server and each particular client box. (Shared long-term between the server and a particular client box).

Short-term material (created during protocol instances):

- W_{CS} – Session key (Rijndael) that the server sends to a client box and is used during one protocol instance.
- n_c – Session identifier (sequence number) generated by the client box. The previous value is remembered by both the server and the client box.

Constants for a given digital content:

- ***file*** – Digital content identifier (or a file name) of the content the client wishes to view.

- ***N_K*** – The number of the content-decryption keys.
- ***I_K*** – Content-decryption key index. ($0 <= I_K <= N_K$)
- ***K_i*** – The ith Rijndael key used for content decryption. If i = 0, it is a CBC initialization vector. If i > 0, it is a Rijndael key.

Messages are self-interpreting as to their type. Here are the message types and the padding symbol:

- ***REQ*** – identifies a request message
- ***SK*** – identifies a message containing the session key
- ***ACK*** – identifies an acknowledgment message
- ***CK*** – identifies a message containing the content key
- ***ABORT*** – identifies an abort message
- ***P*** – Padding bytes (see below)

For padding messages encrypted with RSA we use the *Public-Key Cryptography Standard #1 (PKCS)* which is the standard way of formatting messages to be encrypted with RSA (See *5.2.6 Protocol Design Philosophy*). We use ***P*** to denote the padding bytes {*0x00, 0x02, PS, 0x00*}, where the first byte is 0, the second byte is 2, the next eight bytes are random nonzero bytes, here called *PS*, and the eleventh byte is 0.

5.2.2 Protocol Messages

We first display the protocol exchanges graphically and then explain in English.

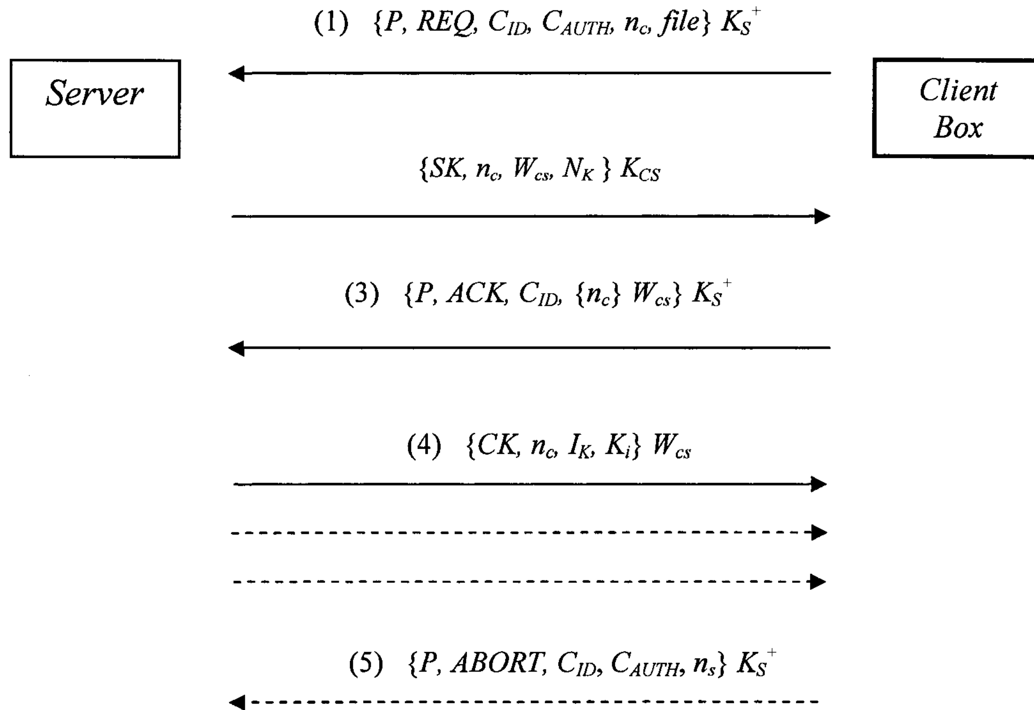

Figure 5.1 Protocol Messages

1. The first message contains P – padding bytes, REQ – a bit pattern that identifies the message as a request message, C_{AUTH} – the client box's authentication string, C_{ID} – the client box's short ID, n_c – a newly generated (by the client box) session identifier to be used for the current protocol instance, and *file* – digital-content identifier. The entire message is encrypted with K_S^+, which is the server's RSA public key.
2. The second message contains SK – a bit pattern that identifies the message as a message containing the session key, n_c – the current session identifier, W_{CS} – the

newly generated (by the server) session key, and N_K – the number of the digital-content-decryption keys for the digital content identified by *file*. The entire message is encrypted by K_{CS} – which is the Rijndael key shared between the server and a particular client box.

3. The third message contains P – padding bytes, ACK – a bit pattern that identifies the message as a session-key acknowledgment message, C_{ID} – the client box's short ID, $\{n_c\}W_{cs}$ – current session identifier encrypted with the newly established session key. The entire message is encrypted with K_S^+, which is the server's RSA public key.
4. The fourth message contains CK – a bit pattern that identifies the message as a message containing one of the content-decryption keys, n_c – the current session identifier, I_K – the index of the content-decryption key in this message, and K_i – the ith Rijndael content-decryption key (if i = 0, it is a CBC initialization vector). The entire message is encrypted with the W_{cs} – the session key established for this protocol instance. Note that there will be as many messages sent as there are Rijndael content-decryption keys for the digital content identified by *file*, plus an additional message containing the CBC initialization vector.
5. The fifth message is not part of the protocol. It is used to assert that not all content-decryption keys were received. It contains P – padding bytes, $ABORT$ – a bit pattern that identifies the message as an abort message, C_{ID} – the client box's short ID, C_{AUTH} – the client box's authentication string and n_c – the current session identifier. The entire message is encrypted with K_S^+, which is the server's RSA public key.

5.2.3 The Protocol in Operation / A State-Transition View

Here we show graphically how the protocol works. We describe the interaction of two finite-state machines, namely the client box and the server. During protocol execution, the server and the relevant client box expect various types of messages while in certain states. Upon receiving an expected message relative to the current state, each machine performs some action and then transitions to another state. Machines ignore unexpected messages. Note that the server may be involved protocol instances with multiple client boxes at the same time. Thus, the server maintains multiple server states, where each server state shows the progression of a protocol instance with respect to one of the registered client boxes. The Figure 5.2 below depicts both the server and some particular client box as they cycle through their states during a protocol instance. A detailed description of the protocol states and actions follows shortly.

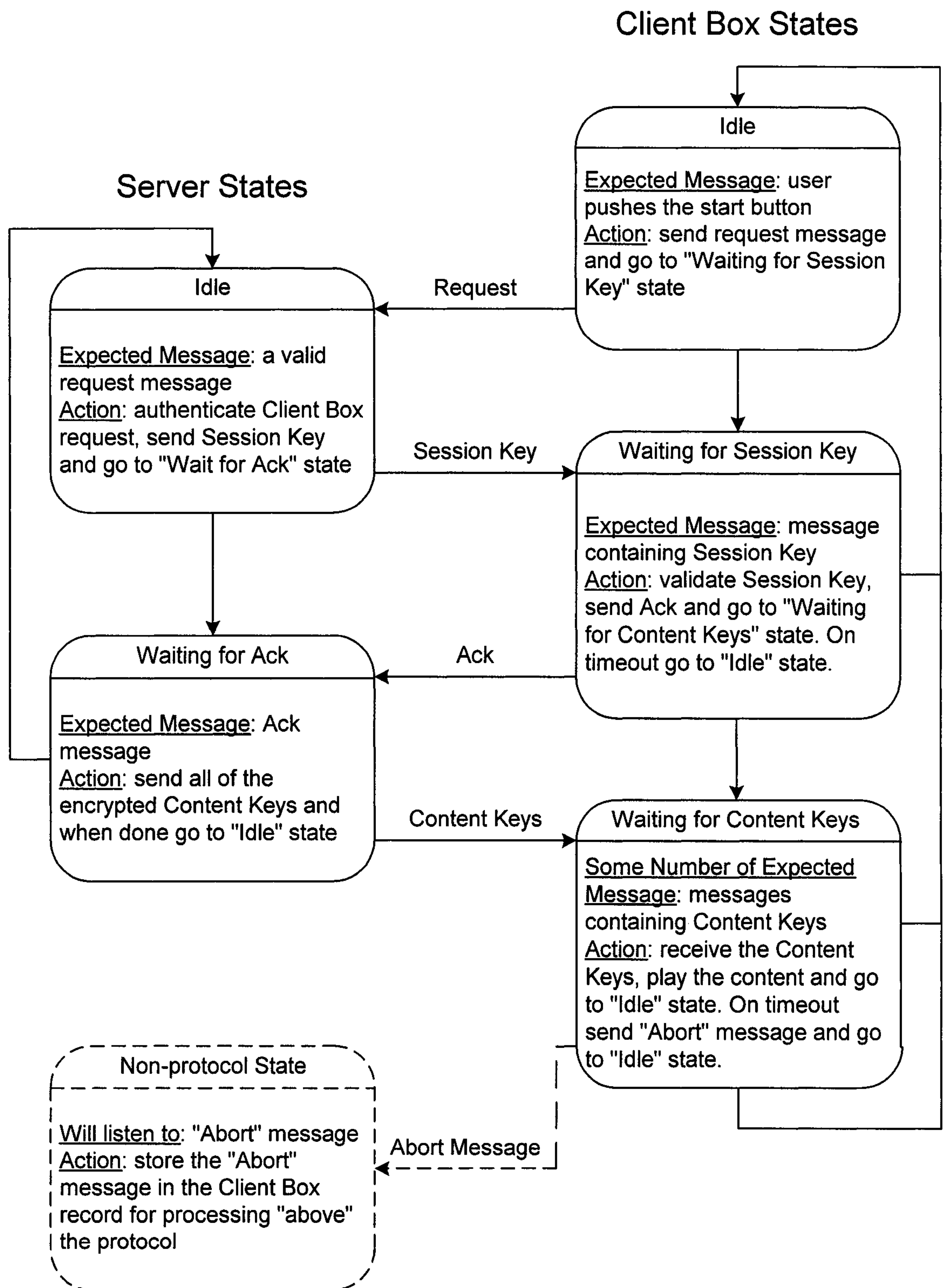

Figure 5.2 Protocol States

5.2.4 Comprehensive Description of the Protocol

This section describes the protocol steps without any justification of the choices we made. The goal of this section is to explain to the reader how the protocol works. A hardware or software developer should be able to implement the protocol simply by reading through this section. In a later section, we present the design philosophy behind the choices that we made. The description is broken into two parts. First, we show the client program represented as a Mealy machine (finite-state machine with both inputs and outputs). The client program is presented by considering each client state in sequence, and showing how the client responds to messages when in that state. Second, we show the server program, also represented as a Mealy machine. The server program may be serving multiple clients simultaneously and may therefore be in different states for different client boxes. The server program is presented by considering each server state in sequence, and showing how the server responds to messages when in that state.

5.2.4.1 The Client Program

Here we describe the actions to be taken by the Client machine. In each state, the client program is listening for a specific message. Only the expected messages will cause the client program to react. Unexpected messages are ignored. After receiving a message, the machine performs some validation to see if that message is expected in this state. If any of the validation steps fail, the client box program discards the message and continues waiting for a valid message. If no valid message arrives within the timeout period, the client box program it returns to the "Idle" state and, in one instance, sends an "Abort" message. In an abort, the session identifier n_s is stored

by the client box and will be increased in future protocol instances; all other protocol data generated or received for this instance of the protocol is erased; the client box returns to its idle state.

Client State: Idle

The only message expected in this state is the user internal message selecting the content to view and pressing the "Start" button. Other messages are ignored. When the user presses the start button, the client box keys up the appropriate encrypted digital content, prepares and sends an authentication/request message to the server containing authentication information, content identification and a client-chosen session identifier. To do this, the client box prepares four entities that will be concatenated, encrypted and sent as one message to the server. Some items, notably the authentication information, are the same between different protocol instances while others are different. We now describe the four entities that the client box prepares for authentication/request message:

1. A 256-bit random identifier C_{AUTH} used to authenticate the client box to the server. This is one of the main long-term secrets stored inside the client box and is used in all the protocol instances of a given client box.
2. A 32-bit identifier C_{ID}, which is a short client box name to allow easier lookups of the client box record on the server side. This is one of the long-term secrets stored inside the client box and is used in all the protocol instances for that client box. (Note: this is an implementation optimization only).

3. A 64-bit session identifier n_s used to identify this protocol instance. This number is generated by the client box for each protocol instance. Session identifiers are random but strictly monotonically increasing. The session identifier is used to prevent self-replay (see *Client State: Waiting for Session Key*) and to authenticate messages from the server that contain either the encrypted Session Key or the encrypted Content Keys (see *Client State: Waiting for Session Key* and *Client State: Waiting for Content Keys*). The client box stores the generated session identifier for this protocol instance. It will be used during this protocol instance and during the generation of a session identifier for the new protocol instance.
4. A 32-bit digital content identifier *file* the user of the client box wishes to view.

All of these four items are concatenated, padded and then encrypted with RSA using the server's shared long-term key K_S^+. The client box then sends the encrypted message to the server and goes to state *Waiting for Session Key*. Symbolically, the message sent is: $\{P, REQ, C_{ID}, C_{AUTH}, n_s, file\}K_S^+$

Client State: Waiting for Session Key

In this state the message client box is expecting is the following message: $\{SK, n_c, W_{CS}, N_K\}K_{CS}$, which should contain the session key for this protocol instance. Here, *SK* is an encoding that this is a Session Key message and the other symbols have been defined. When the message is received, the client box uses the long-term Rijndael key K_{CS} and the long-term CBC initialization vector I_{CS}, both of which are shared

between the server and a particular client box to decrypt the received message. When the message is decrypted, three entities are extracted, namely the session identifier n_c, the session key W_{cs} and the number of the content-decryption keys N_K that it should expect. The client box then performs the following actions:

1. The client box checks the session identifier n_c to guarantee that this message belongs to the session instance it expects.
2. The client box saves the session key W_{cs} to be used for the current instance of the protocol. Specifically, the session key is used by the server to encrypt and by the client box to decrypt the content-decryption keys, themselves entirely random.
3. The client box saves the number of the content-decryption keys N_f that it should expect from the server for the requested digital content. These will arrive in encrypted form and will be decrypted using the session key.

After successfully verifying the session identifier n_c, and storing the session key W_{cs} and the number of the expected content-decryption keys N_K, the client box prepares an acknowledgment message for the server. The client box takes the session identifier n_c and encrypts it with the newly received session key W_{CS}. It then takes its short identifier C_{ID}, the encrypted session identifier $\{n_c\}$ W_{cs}, pads the message with PKCS, encrypts it with the server's RSA encryption key K_S^+ and sends the message to the server. The first byte of the data part is an encoding, namely *ACK*, that identifies the given message as an acknowledgment message. After sending the

acknowledgment message the client box goes into state *Expecting Content Keys*. Symbolically, the message sent is: $\{P, ACK, C_{ID}, \{n_c\} W_{cs}\} K_S^+$

Client State: Waiting for Content Keys

In this state the client box is waiting for multiple messages from the server, each of which will contain one of the 256-bit content-decryptions keys K_i encrypted with W_{cs}. Symbolically, the expected message is the following: $\{CK, n_c, I_K, K_i\} W_{CS}$. The number of messages expected by the client box in this state is just the number of content-decryption keys N_K. Here, CK is an encoding that identifies this as a Content Key message. When a message is received, the client box decrypts it using the session key W_{cs} established for this session between the server and the client box. When the message is decrypted, three entities are extracted, namely the session identifier n_c, the ordinal number (index) of the content-decryption key I_K and the content-decryption key K_i. For every message containing a content-decryption key, the client box performs the following actions:

1. The client box checks the session identifier n_c to guarantee that this message belongs to the session instance we expect.
2. The client box saves the index I_K of the content-decryption key, so that we know which order to apply the keys.
3. The client box saves that particular content-decryption key K_i.

Once all of these messages are received, the client box is ready to play the content. The client box will first delete the session key W_{cs} and then will play the digital

content. Content-decryption keys will be flushed as soon as their use is complete. Although there is a tamper-proof implementation, this further minimizes the window of vulnerability.

If the client box does not receive all the content-decryption keys in a timely fashion it will send the "Abort" message to the server, which symbolically is: $\{ABORT, C_{ID}, C_{AUTH}, n_s\}K_S^+$. Here, *ABORT* is an encoding that identifies this as an "Abort" message. After sending the "Abort" message the client box will erase the dynamic protocol data and will transition to the "Idle" state. (Thus, the box may abort the protocol with or without sending an "Abort" message).

5.2.4.2 The Server Program

Here we describe the actions that are taken by the server program. As before, the server has states, but these are per registered client box. Thus, the server is a collection of Mealy machines. When the server receives a message from a client box it must first determine from which client box the message is coming in order to determine what state it is in with respect to that client box. Therefore, some processing is required on the server's part before it knows how to react to the client box message it just received. For a given client box, the server will be expecting either a request or an acknowledgment message. The server may also receive an optional "Abort" message from the client box it the latter did not succeed in completing the protocol instance. Note that "Abort" messages are handled at a higher level than the protocol itself; strictly speaking they are transparent to the protocol.

We have seen that the client box expects messages of various kinds from the server; having distinct client states allows the client box to ignore unexpected messages. The situation in the server is the same. However, since the Server may receive messages from any number of client boxes, it needs one state per client box. Unexpected messages from a given client box are ignored; they are treated as unexpected relative to the stored state of a specific protocol instance (or absence of such) of a specific client box. In other words, the server has one state per client box. The server maintains state information for each client box including information about any protocol instance it is engaged in. The server program listens for either a request message for a protocol instance to be initiated or an acknowledgement message that belongs to some protocol instance that is underway. When a valid request message arrives, the server acts upon it by sending the session key to the client box program, and when a valid session-key acknowledgment message arrives, the server acts upon it by sending the content-decryption keys to the client box program.

Even after sending all content-decryption keys, the server does not know immediately whether a protocol instance has completed successfully. Nonetheless, it behaves as if this were the case as soon as it has sent the last content-decryption key. If the client box times out waiting for the content-decryption keys it will send an "Abort" message. If the server receives an "Abort" message, it makes a note of this fact. Action based on the "Abort" message is decided by the higher administrative layers and is not part of the protocol per se.

All the incoming messages are encrypted with the same RSA encryption key K_S^+, which is not publicly known but is a long-term shared secret shared between the server and all the client boxes. Once the server receives a message, it uses its RSA decryption key K_S^- to decrypt the message. When the message is decrypted, the server checks which client box the message is coming from, fetches the record it maintains about this client box, and verifies whether the message is expected given the current server state with respect to this client box.

Server State: Idle (Waiting for Request Message)

In this state the server is expecting a request message from a registered client box. Symbolically the expected message is the following: $\{P, REQ, C_{ID}, C_{AUTH}, n_s, file\}K_S^+$. As mentioned earlier, *REQ* is an encoding that identifies this as a request message.

Briefly, in order to for a request message to be authenticated, the client box identifier C_{AUTH} must match the one the server has on record and the session identifier must be greater than the one stored in the client box record, which is maintained by the server and contains all the information the server has about that client box. In detail, the server performs the following actions:

1. The server uses the short identifier C_{ID} to fetch the record of a particular client box.

2. If the record exists the server goes on to C_{AUTH}. The server compares the C_{AUTH} it received with the one in the client box record. If they match the server goes on to the session identifier n_s.
3. The server's client box record, amongst other things, contains the session identifier used during the last initiated protocol instance between the server and that client box. The server will verify that the current session identifier is greater than the one stored in the client box record.
4. Given a C_{AUTH} match, if the session identifier n_s is greater than the one stored in the record, the identity claim of the request has been authenticated and we know that this is a new request from that client box. The server then stores the current session identifier n_s and the current digital content identifier *file* in the client box record.
5. The server fetches the number of the decryption keys N_K for the requested digital content.
6. The server generates a fresh "true random" session key W_{CS} that will be used to encrypt all digital content keys it will subsequently send to the client box.
7. The server then concatenates the session identifier n_c, the newly generated session key W_{CS} and the number of the decryption keys N_K. The concatenation of these values is encrypted with the long-term Rijndael key K_{CS} and the long-term CBC initialization vector I_{CS} shared between the server and this client box.
8. The server sends the encrypted message to the client box.

After sending the session key message the server goes into state *Waiting for Acknowledgment Message* with respect to this particular client box. Symbolically, the message sent is: $\{SK, n_c, W_{CS}, N_K\}K_{CS}$

Server State: Waiting for Acknowledgment Message

In this state the server is expecting an acknowledgment message. Symbolically the expected message is the following: $\{P, ACK, C_{ID}, \{n_c\}W_{cs}\}K_S^+$.

After decrypting the message the server performs the following actions:

1. The server uses the short identifier C_{ID} to fetch the record of a particular client box.
2. The server verifies whether the client box possesses the newly established session key W_{CS} by decrypting the current session identifier n_c that the client box encrypted with the session key W_{CS}.

After the server has validated the acknowledgement message, it sends the content-decryption keys encrypted with the session key established for this protocol instance. The number of the digital content-decryption keys the client box should expect was sent to it previously in the message containing the session key. Each message the server sends consists of the concatenation of the session identifier n_c, the index I_K of the current content-decryption key and the content-decryption key itself K_f. Each message in this sequence is encrypted with new newly established session key W_{CS} and symbolically is: $\{CK, n_c, N_{Kf}, K_{fs}\}W_{CS}$. As an added security layer, we send the

keys in the reverse order, i.e., from K_n to K_0, thus sending the CBC initialization vector I_{CS} (K_0) last. The CBC initialization vector is the first piece of key material used when decrypting the digital content. Once the content-decryption keys have been sent, the server regards this session as having completed successfully and goes into state *(Idle: Waiting for Request Message)* with respect to this particular client box.

Content Key Abort Message (Not state specific)

If the client box does not receive all of the content-decryption keys, it will time out and sent an "Abort" message. Although this message is received by the server and recorded, it is transparent to the protocol and will be handled by higher administrative layers. Symbolically the abort message is: $\{ABORT, C_{ID}, C_{AUTH}, n_s\}K_S^+$. Note that this message will be accepted in any state, given only that the content-decryption keys were in fact sent.

5.3 Protocol Security

Although the ways in which the adversary can interact with a protocol run are essentially infinite, we would like to demonstrate that our protocol is not susceptible to the most commonly encountered threats to cryptographic protocols. Each attack has already been briefly described in Chapter 2.

The *Eavesdropping Attack*, i.e., listening in, is usually circumvented by using encryption. All of our protocol messages are encrypted and therefore are invulnerable to the Eavesdropping Attack.

In the *Replay Attack*, an adversary (here, the untrusted client box user) can record all the message traffic from a successful protocol run that was used previously to display digital content 'A'. The adversary can then initiate a new instance of the protocol by requesting to display digital content 'A' again, but instead of communicating with the server, the user can simulate server responses by re-playing pre-recorded server messages from the previous successful protocol run. In order to prevent self-replay from happening, we require that, for every new request, the protocol running on the client box generates a fresh session identifier and stores it for the duration of the protocol run. When the client box receives the message with the session key, it will first check the session identifier and then proceed with the protocol only if the session identifier is correct, thus preventing self-replay and protecting from the *Replay Attack.*

The *Modification Attack* involves modification of a message or message fields. We have a number of integrity message checks. In particular, all our messages contain the session identifier, which is verified after the message is decrypted and before the message is processed, thus confirming that the message was not tampered with.

The *Reflection Attack* requires that parallel runs of the same protocol are allowed. We only allow one instance of the protocol at a time to be run from a particular box.

Remember that the program in the client box is hard-wired. Instances from distinct client boxes are easily kept separate by the server, because of the unique client box identifier.

Within the scope of this thesis, there is no comprehensive solution to counter a *Denial of Service* attack. If our connectivity is destroyed, this is outside the range of threats we are prepared to deal with. On the other hand, the protocol does discard ill-formed or spurious messages so only denial of connectivity or flooding is dangerous.

The *Typing Attack* (confusion as to type) exploits the fact that, although the protocol elements are clearly distinct when written on a piece of paper, in practice the principal receives a bit string and has to interpret it. In our case each message contains a message type identifier, so that there can be no confusion (ambiguity) about the message type.

The ciphertext-only *Cryptanalysis Attack* is only possible if some underlying plaintext redundancy exists and has been characterized. Normally, you massage the ciphertext by various cryptographic transformations in the hopes of moving it closer to the plaintext, which you sense by seeing if you have decreased the entropy. In our protocol, all the plaintext – the content decryption keys and the session keys – that the *server* sends to the client box is perfectly random. Digital content keys are generated prior to any protocol instance on a machine that has a source of true randomness. Session keys are generated entirely by the server, which has access to a source of true randomness. Therefore the entropy of the plaintext sent by the *server* is maximal. Although not all plaintext set by the *client box* is perfectly random or pseudorandom, our use of "hidden" RSA (that is, no

exposure of any keying material) limits any vulnerability. The client identifier is perfectly random, the session identifier is pseudo-random and only the content identifier is fixed.

The *Certificate Manipulation Attack* does not apply to our protocol since we do not use public certificates to identify the clients, but use built-in ids for that purpose. Certificates are required when one protocol participant must prove its identity to another. In our protocol, each client box has a built-in secret ID with which it identifies itself to the server. The client box operator has no access to the value of this ID. It is sent to the server encrypted with the server's public encryption key. Only the server has the private decryption key, thus only the server is able to decrypt the message with client's secret ID.

The *Protocol Interaction Attack* in general can be effective if the long-term keys originally created to be used for a single protocol and are then used in other protocols. Since we only have one protocol in our security architecture this attack does not apply.

6 Conclusion

In this thesis we formulated a new DRM problem and then attempted to solve it. Although our primary motivation was the DRM issues in delivering high-value digital content such as first-run movies, the security architecture may be generic enough to be useful in other areas, as we demonstrated in the introduction by the military example. By changing the conventional trust model we created a new set of assumptions. In the conventional trust model, as shown in **Figure 6.1,** there are two users that want to communicate with each other. Both of the users are trusted. Both of them have access to cryptographic keys and plaintext content. In particular, each will have access to shared cryptographic keys and will be in control (as a programmer) of the protocol steps.

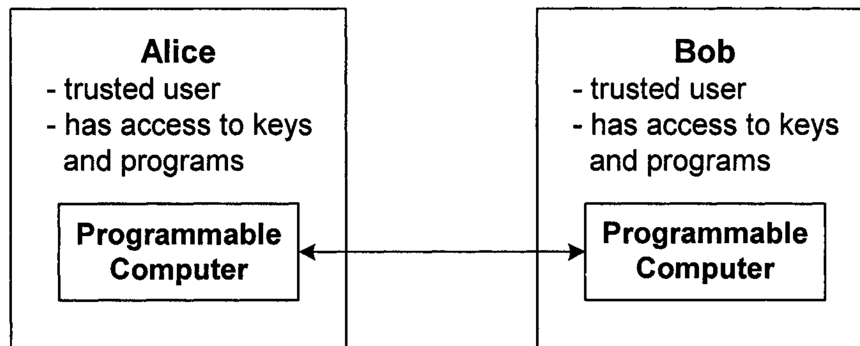

Figure 6.1 Conventional Trust Model (For Communicating)

In our unconventional trust model, as shown in **Figure 6.2**, we have one trusted user (namely, the server operator) and multiple untrusted users (namely, each of the client box users).

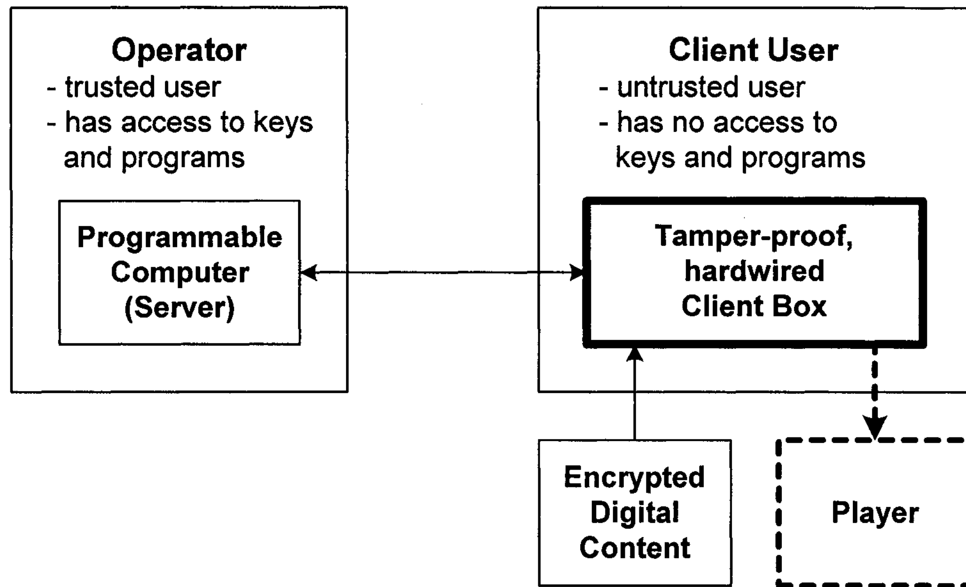

Figure 6.2 Unconventional Trust Model (For Capability Passing)

6.1 Design Philosophy

The DRM problem we wanted to solve consisted of two parts. The first one was "How can we achieve a separation of concerns so that delivery of encrypted digital content is orthogonal to having the capability to display it, which requires decrypting it?" The second one was: "How do we deliver the capability to display digital content?"

This gave rise to numerous sub-problems. We had to find an encryption mechanism that was robust and fast to decrypt. The fact that it had to be fast right away led us to choose a symmetric algorithm. We decided to look at the available symmetric algorithms and see which one would fit our needs. The first and most important characteristic was the security of the algorithm. The second one was the efficiency of the algorithm. Naturally,

we started looking at the protocols submitted for the Advanced Encryption Standard (AES). We chose Rijndael, which is the current AES.

One of the conditions for an AES candidate was that the block size must be 256 bits and the key size must be 128, 192, or 256 bits. Although some of the submitted protocols were rated as more secure than Rijndael, we still chose Rijndael because of its extensibility. That is, Rijndael can grow the block size and key size to any bit value that is a multiple of 32. As we showed in Chapter 3, there is no known attack on Rijndael that is much better than the brute-force attack. We chose to use a 256-bit block and a 256-bit key in our security architecture. As we see from Chapter 3, the closest configuration that was analyzed during the competition to see if it could be broken was a 128-bit block with 256-bit key and a reduced number of rounds (9 as opposed to 14). The best short cut found a marginally better attack than the brute-force attack. That is, in order to break the encrypted message, instead of 2^{256} operations we would need 2^{224}.

If we plot any reasonable utility function of the value of cracking the movie after time t_c, we see that the cost of doing so vastly exceeds the utility no matter how much computer power we choose to employ. We ran a little test on a PC with 2.8GHz CPU and 1GB of RAM to estimate the time required to try every key in the 256-bit key space. It took on average 70 seconds to try every key in a 2^{20} key space. If we multiply the 70 seconds by $2^{(256-20)}$ we get $2.45 * 10^{65}$ years. Generally, a movie is most valuable within 1 year of its release, giving essentially a step function for the utility function. In order to try every key in a 256-bit key space within one year, we would need $2.45 * 10^{65}$ PCs whose value

is obviously greater than the movie itself. A speedup by a factor of 100 or even a 1000 would make no difference, e.g., if we were to use Field-Programmable Gate Arrays (FPGAs). The cost of obtaining and running any configuration whose time to solution is sufficiently small is obviously greater than the value obtained.

Rijndael is also very efficient on the variety of CPUs, thus giving us the choice of using either a merchant ("commodity") processor or a custom Application-Specific Integrated Circuit (ASIC) inside the client box.

Designing a way to deliver the capability to diplay digital content was a much harder task. We had to find a way to deliver the actual capability and protect it on the client side. We looked at the tamper-proof hardware available today and concluded that it would be possible to design hardware that would meet our requirements. We have discussed the IBM example to demonstrate this. As we mentioned earlier, design of such hardware is outside of the scope of this thesis. However, we wanted to touch on some of the design choices we made for the tamper-proof hardware box.

The main contribution of this thesis is the identification of a new DRM problem characterized by its unconventional trust model together with a key transport protocol we designed to transmit the capability to display high-value digital content, in a controlled fashion, given our assumptions about the existence of a tamper-proof box with a hard-wired program and shared secrets that were not really shared.

The question we set out to answer was, is it possible to combine various forms of well-known security components in a security architecture in such a way that it would be so manifestly robust that content providers would entrust us with delivering their truly high-value digital content? The design philosophy we followed was to 1) give the user a ***white box,*** unlike Sony BMG, 2) deliver capability rather than content, and 3) use protocols for authentication and key transport.

The *Request Messages* – messages sent to the server – are encrypted using the RSA algorithm. We wanted to use one key to encrypt all messages sent to the server, so that we would not have to send a client box identifier in plaintext along with the encrypted message – in order for the server to know which key to use to decrypt the incoming message. Using a symmetric key is slightly more dangerous, because if one client box is compromised then the adversary can learn the secret ids of any client box by decrypting recorded requests to play digital content and thus impersonate any client box. Using an asymmetric key (RSA in our case) is safer, because discovering the encryption key does not automatically give the adversary the decryption key.

Each message encrypted with RSA is padded, using Public-Key Cryptography Standard #1, to improve message security. Also, since both the encryption key (exponent) and the modulus for RSA are unavailable, the standard RSA cryptanalysis techniques do not apply.

To avoid any ambiguity as to the message type, each message contains a message type identifier. These identifiers are perfectly random bit-strings, so they do not increase the entropy of the plaintext.

Each message contains a session identifier. The session identifier is primarily used to prevent self-replay by the client box user. It is also used to filter the messages the client box allegedly receives from the server. If a client box receives a message that is not from the server, it can easily reject this message by passing it through the decryption mechanism and failing to verify the session identifier. The unique session identifier is also used by the server for accounting purposes, to keep track of the protocol runs initiated by a particular client box.

The client box is securely identified by its long random identification string. For efficiency reasons, we also use a short identification string. By providing the client box short identification string, we allow the server to do faster lookups of the client box record.

The *Session Key Message* – the message the server sends to a client box containing the session key for a particular protocol instance – is encrypted using a symmetric key shared between the server and a particular client box. Most of the plaintext content of the message is random and therefore has near maximal entropy, thus making ciphertext-only Cryptanalysis Attacks difficult if not impossible.

One of the items the session key message contains is the session identifier. It is used by the hardwired client box programs to guard against self-replay. Without it, the client box user could record all the traffic from the server during a valid protocol execution and then play the content multiple times without the server being involved. The client box user can request to play the same content and then intercept all the outgoing client box messages to the server, and thus hiding all activity from the server, and then feed the box the pre-recorded server responses from a successfully completed protocol instance. With the session id freshly generated and stored inside the client box for the duration of a particular protocol instance this is impossible. The hardwired client program checks whether the message containing the session key belong to the current protocol instance and ignores the message if it is not, i.e., when the session identifier does not match the one saved by the client box.

By sending the *Acknowledgment Message* the client box confirms that it possesses the correct session key for this protocol instance. It does so by encrypting the current session identifier using the newly established session key and sending to the server inside the acknowledgement message. When the server decrypts the session identifier using the session key and verifies that the session identifier is current, it knows that it could have only be encrypted with the session key by the client box. Session key confirmation is done without the actual key being sent by the client box.

The *Content Key Messages* are encrypted with the newly established session key. Each digital content entity is typically encrypted with several Rijndael keys – one distinct key

per segment. Content key messages also contain a content key index, since these messages may arrive out of order. Only after all of the content decryption keys arrive is the client box able to use any of them.

The *Abort Message* will be sent any time a client box times out before receiving all the content decryption keys. Still, this message is "optional" in the sense that it does not lead to any server protocol options (other than reporting to an administrator). If the client box does not receive all of the content keys, say, due to network problems, it can timeout and send an abort message to the server. If the server receives an abort message, it is left to up to the administrator to deal with the fact that the protocol instance has failed. Normally, an abort message is evidence that the protocol did not complete successfully.

6.2 Related Work

We did not find someone working on precisely the same problem. Much DRM work is kept secret by a foolish belief in security by obscurity. However, there is a wide range of DRM problems that people are actively working on. Some of these DRM problems are more general than ours, e.g., "Can I play the digital content on my PC and on my iPod in the car?" Most of the problems are members of the copy-protection family rather than members of the capability-to-display family. Most often, solutions to these problems are described at a very high level. Not many details are available about the protocols that are used, which makes it impossible to do a one-to-one technical comparison with our solution. We will list some DRM problems and (sketches of) solutions of some other people.

Sun Microsystems has initiated an open-source community project to develop a royalty-free digital rights management standard. It is planning to share the entirety of its internal Sun Labs Project DReaM (DRM/everywhere available) with the community. Sun believes that a federated DRM solution must be built by the community for the community. We mention this only to show that others are working on somewhat related problems, not because we think the Sun initiative will amount to anything.

Project DReaM includes an interoperable DRM architecture called DRM-OPERA as well as some technology components for digital video management and distribution. OPERA achieves interoperability among DRM systems essentially by reducing DRM licenses to the lowest common denominator of authenticating users only and providing "play once" as an atomic licensing term that all DRM systems can understand and support. OPERA's approach does not work well with rights that differ from "play", such as rights to copy, move, and even render content in other ways, like "print". If a single DRM system is to control more complex rights expressions, like "play n times" or "play for a week", then it will not be able to support rights to export content to other systems under terms other than the simple "play once" [16].

Microsoft's DRM solution for Windows Media - Windows Media Rights Manager – lets content providers deliver digital content, such as music, videos, etc., over the Internet in a protected, encrypted file format. Encrypted digital content entities are packaged with extra information about the media file in a packaged file. The extra information includes

among other things the URL where the license can be obtained. The encrypted license contains the content key and is distributed separately. The protected packaged files can be obtained in a number of ways, including Internet download, CD distribution, e-mail, etc. In order to play a packaged digital media file, the consumer must first acquire a license. The license contains the content key that unlocks the media file as well as the rights (or rules) that govern the use of the digital media file. Possible rights include the number of times the file can be played, which devices the file can be played on, when the user can start playing the file and when this right expires, whether the file can be burned onto a CD, etc. [17].

If we compare our architecture to the Windows Media Rights Manager, we see that Microsoft's way of content distribution in a protected form is somewhat similar to ours. We both have the digital content distributed in an encrypted form from a variety of sources, including Internet download, CD distribution, e-mail, etc. In Microsoft's scheme the user requests a license on-line. The license the user receives contains the key to decrypt the digital content. A programmable computer unpacks the license, retrieves the key, decrypts and plays the digital content. Microsoft has to use an elaborate scheme to obscure what is happening behind the scenes in order to protect plaintext digital content and the digital content keys. There are a number of exploits already available that successfully attack and defeat this DRM scheme. There are also exploits that simply capture the plaintext digital content on its way to the sound card, in the case where the digital content is music. In our architecture, the user is not using a programmable computer but instead a tamper-proof client box very much like set-top box. The user has

no access to plaintext digital content or digital content keys. Because of the use of the tamper-proof box, we do not need to employ security by obscurity, which seems to be the only solution in the case of the programmable computers, to hide the secrets from the user. We know that any attempt to defeat the tamper-proof box will result in total erasure out of all the short-term and long-term secrets. Also, since we assume total security on the client box side, we can have a simple protocol with, it appears, no possibility of successful protocol attack.

The fundamental difference between us and Microsoft is that they make security depend on the alleged difficulty of reverse engineering (the major false assumption underlying conventional DRM) and we do not.

A European project TIRAMISU (The Innovative Rights and Access Management Inter-platform Solution) is geared towards digital video in home entertainment networks. It is essentially a SmartCard-based design intended to create a smooth transition for legacy conditional access (CA) players. TIRAMSU calls for authentication of devices and domains but not users. The TIRAMISU project enables scenarios, in which users are able to get media content (in protected form) from various sources. In order to access the content the user must have a license issued by the rights owner for each content instance. These licenses are stored on the SmartCards and are portable between devices [18].

In TIRAMISU's solution, as well as in ours, the protected digital content is delivered in an encrypted form. In TIRAMISU's case the capability is distributed on a SmartCard

that the user has to physically obtain. In our case the capability is securely delivered on-line. In TIRAMISU's solution content decryption is done in a set-top box that is not tamper-proof. Extracting the plaintext digital content from an unprotected set-top box is difficult but not impossible. The difficulty of extracting it from a set-top box is probably enough to discourage a user from extracting content that can easily be purchased on a DVD for $20. However, it is not enough discourage a user from extracting a first-run movie worth millions of dollars. For this we need a tamper-proof hardware box capable of instantaneously and securely deleting all the secrets upon detecting tampering.

We close this section with an incidental remark. One of the problems that our security architecture solves is the issue of movie piracy or more precisely, protecting digital bit streams from being pirated up to the moment of display. Of course, there are complimentary aspects of protection here. According to New York Times, Hollywood movie studious are frustrated with what they view as laziness or reluctance on the part of consumer electronics and information technologies industries to invest in anti-piracy technology. Six movie studios have partnered on a research laboratory (MovieLabs) to develop new techniques to thwart film pirates. Some of the initial investigations will include methods of disrupting the recording of movies inside cinemas by camcorders, preventing home and personal digital networks from being hacked while allowing consumers to send content to multiple TVs without being overcharged, detecting unauthorized content sharing on peer-to-peer networks, spotting and impeding illegal file transfers on campus and business networks, connecting senders and receivers of films

relayed over the Internet to geographic and political territories, watchdogging the distribution of movies, and curbing license agreement violations [19].

6.3 Future Work

There are several open questions. Perhaps the basic question is: "How can this security architecture be extended to other DRM problems?" Our current architecture only supports the "play once" right. It is trivial to see how to extend it to very similar but more complex rights such as "play n times" or "play for a week". The real question therefore is how to add flexibility so that the DRM system controls the acquisition of a particular capability out of a much wider set of capabilities. This of course requires the construction of a more general "player". Essentially, the display system needs to become more like a general purpose information appliance.

The future research program with the most promise and interest is to move into the more general area of *Information Rights Management* (IRM), where users are permitted some subset of "access" rights among all the access rights that might be granted for a particular information object, to accomplish similar things to what an operating system accomplishes when it uses an access control list in a file system. For example, if the digital content were an e-mail, the full set of rights might include displaying, forwarding, printing, saving, etc., and the IRM protocol would allow the user to acquire a proper subset of those rights.

Another, unrelated open problem concerns not so much extending our single-capability DRM system into a multi-capability IRM system (as discussed above), but rather finding a good solution to the "last-mile" problem in the original DRM problem. Essentially, at some point the plaintext has to go from the box to the display unit. Thus, it would leave the tamper-proof vault. We need to prevent theft of the digital content while it is being transmitted from the box to the display unit. Possible solutions include quantum-cryptographic techniques to detect signal capture (and shut the system down) during such transmission or possibly incorporating both box and display unit into a single tamper-proof vault.

In this thesis we have attempted to find a solution to a special-purpose DRM problem that is perhaps worthy enough to deserve a solution optimized precisely for this one problem. The problem is, deliver the capability for one-time display of a very high-value digital content. The content in this case may be a digital media file or a special program. We think it may potentially have other DRM as well as military applications. Since much of what the military does is classified, we cannot provide any real military scenarios where our solution can be applied, nor can we compare our solution to any of the solutions available for the military. Therefore, we have mostly treated our solution as a pure DRM solution.

In all honesty, considering the interest in multilevel security in most classified shops, the real military value probably lies in an expanded, opened-up version of the architecture, in

which capabilities for some subset of a multitude of "access operations" would be securely distributed to different users (with different clearances) of information objects.

In short, we suspect that the important next-generation civilian and military IRM problems are one and the same.

In conclusion, we strongly recommend the vital and underexplored problem of Information Rights Management (IRM), where IRM is rights management of (sensitive) distributed shared information. This involves a shift of focus from piracy to privacy (civil liberties). It is required for future databases containing such things as health or counterterrorism information (indeed, to civilize many future federal data activities). Here are two sample IRM problems and an IRM recommendation: 1) who can access and use sensitive information?, 2) can sensitive information be printed, forwarded, or copied by unauthorized people?, and 3) IRM policies & guidelines and IRM technologies should be developed in tandem – after all, they are both in their infancy.

References

[1] Joan Daemen and Vincent Rijmen, *The Design of Rijndael,* Springer, 2002

[2] Bruce Schneier, *Applied Cryptography,* John Wiley & Sons, Inc, 1996

[3] Colin Boyd & Anish Mathuria, *Protocols for Authentication and Key Establishment,* Springer, 2003

[4] Michael Luby, *Pseudorandomness and Cryptographic Applications,* Princeton Computer Science Notes, 1996

[5] Aviel D. Rubin, *White-Hat Security Arsenal,* Addison-Wesley, 2001

[6] Alfred J. Menezes, Paul V. van Oorschot and Scott A. Vanstone, *Handbook of Applied Cryptography,* CRC Press, 1997

[7] William F. Friedman, *The Classic Elements of Cryptanalysis,* Aegean Park Press, 1976

[8] Douglas R. Stinson, *Cryptography: Theory and Practice,* CRC Press, 1995

[9] Douglas R. Stinson, *Cryptography: Theory and Practice, Second Edition,* Chapman & Hall/CRC Press, 2002

[10] NIST, *Report on the Development of the Advanced Encryption Standard (AES)*, October 2, 2000

[11] Peter Gutmann, *Secure Deletion of Data from Magnetic and Solid-State Memory,* Sixth USENIX Security Proceedings, July 22-25, 1996

[12] Sean W. Smith, Steve Weingart, *Building a high-performance, programmable secure coprocessor,* Computer Networks 31 (1999) 831-860

[13] Bennet Yee (Microsoft Corporation), J.D. Tygar (Carnegie Mellon University), *Secure Coprocessors in Electronic Commerce Applications*, (http://www.cs.berkeley.edu/~tygar/papers/Secure_coprocessors_in_e-comm.pdf)

[14] FIPS PUB 140-2, *Security Requirements for Cryptographic Modules,* May 25, 2001

[15] Niels Ferguson, John Kelsey, Bruce Schneier (Counterpane Internet Security, Inc., CA, USA), Stefan Lucks (University of Mannheim, Mannheim, Germany), Mike Stay (AccessData Corp, UT, USA), David Wagner (University of California Berkeley, CA, USA), and Doug Whiting (Hi/fn, Inc, CA, USA), *Improved Cryptanalysis of Rijndael*, (http://www.schneier.com/paper-rijndael.html)

[16] Bill Rosenblatt, *Sun's Open-Source DReaM*, September 1, 2005 (http://www.drmwatch.com/standards/article.php/3531651)

[17] Microsoft Corporation, *Windows Media DRM 10*, (http://www.microsoft.com/windows/windowsmedia/drm/default.aspx)

[18] TIRAMISU - The Innovative Rights and Access Management Inter-platform Solution, (http://www.tiramisu-project.org)

[19] Ed Dawson, Andrew Clark and Mark Looi, *Key management in a non-trusted distributed environment*, Elsevier Science B.V., March 1999

[20] Martin Abadi, Roger Needham, *Prudent Engineering Practice for Cryptographic Protocols*, Digital Equipment Corporation, November 1, 1995

[21] Sreekanth Malladi, Jim Alves-Foss, Robert B. Heckendorn, *On Preventing Replay Attacks on Security Protocols*, Center for Secure and Dependable Systems, Department of Computer Science, University of Idaho, Moscow, ID 83844 USA

[22] Tuomas Aura, *Strategies against Replay Attacks*, Digital Systems Laboratory, Helsinki University of Technology, Finland, 1997

[23] Lawrence C. Paulson, *Proving Security Protocols Correct*, Computer Laboratory, University of Cambridge, England

[24] The New York Times (David M. Halbfinger), *Hollywood Unites in the Battle to Wipe Out Movie Pirates,* September 19, 2005, (http://www.nytimes.com/2005/09/19/business/19film.html)