Asymmetric Reversible Parametric Sequences Approach to Design a Multi-Key Secure Multimedia Proxy: Theory, Design and Implementation

Yeung Siu Fung

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Department of Computer Science & Engineering

June, 2003

Asymmetric Reversible Parametric Sequences Approach to Design a Multi-Key Secure Multimedia Proxy: Theory, Design and Implementation

submitted by

Yeung Siu Fung

for the degree of Master of Philosophy
at the Chinese University of Hong Kong

Abstract

Because of the limited server and network capacities for streaming applications, *multimedia proxies* are commonly used to cache multimedia objects such that, by accessing nearby proxies, clients can enjoy smaller start-up latencies and receive a better quality-of-service (QoS) guarantee (e.g, reduced packet loss and delay jitters for their multimedia requests). However, the use of multimedia proxies increases the risk that multimedia data are exposed to unauthorized access by intruders. In this paper, we present a framework for implementing a secure multimedia proxy or, more generally, a secure proxy architecture for audio and video streaming applications. The framework employs a notion of *asymmetric reversible parametric sequence* to provide the following security properties: (i) data confidentiality during transmission, (ii) end-to-end data confidentiality, (iii) data confidentiality against proxy intruders, and (iv) data confidentiality against member collusion. Our framework is grounded on a *multi-key RSA* technique such that system resilience against attacks is provably strong given standard computability assumptions. We also propose the

use of a set of *encryption configuration parameters* (ECP) to trade off proxy encryption throughput against the presentation quality of audio/video obtained by unauthorized parties. Implementation results show that we can *simultaneously* achieve high encryption throughput and extremely low audio/video quality (in terms of audio fidelity, and peak signal to noise ratio and visual quality of decoded video frames) during unauthorized access.

摘要

由於受到伺服器和網絡負荷上限的限制，代理伺服器常常被利用於串流式多媒體播放的應用當中。代理伺服器會把多媒體物件暫存起來，藉由從最近的一個代理伺服器中存取資料，用戶端可以以一個較快的起始速度（較少的網絡延遲）和較穩定的質素（較少的資料封包流失）去享用串流式多媒體播放的服務。可是，代理伺服器的使用提高了入侵者對多媒體資料的未受權存取的可能性。在這份論文中，我們展示一個體現出安全性的多媒體代理伺服器的架構。或者我們更普及化地說，這是一個爲串流式的影像和聲音播放的應用系統而設計的安全代理伺服器的架構。這個架構採用了一個名爲"**asymmetric reversible parametric sequence**"的概念，藉以提供以下各樣特性：

1. 資料傳輸時的安全性
2. 端對端的資料安全性
3. 資料對於代理伺服器入侵者的安全性
4. 資料對於會員間串謀攻擊的安全性

我們的架構是建基於"**multi-key RSA**"的技術之上，因此基於一般的計算機可算性假設，我們的架構是非常有能力抵禦入侵者的攻擊。我們亦都提議使用一套名爲"**encryption configuration parameters (ECP)**"的參數在資料加密吞吐量和已加密資料的破壞程度之間作出權衡。具體研究成果顯示出我們的架構能夠**同時**間達到高度的資料加密吞吐量和極高的已加密資料破壞程度（以未經受權者所能獲得的聲音質素、高峯訊號雜訊比、及解碼後的畫面質素來衡量。）

Acknowledgment

I owe great thanks to my advisor, Prof. John C.S. Lui. I appreciate for his concrete contributions to my work and to my development, and for the enlightment he gave to me on various fields such as presentation skills. I am very grateful that he let me pursue my interests in doing my reserach, from decision of research topic to selection of project extensions. I'm sure that I cannot finish this thesis without his help and guidance, so I must say thank you to him again.

I would also like to thank students in our research study group; thanks Patrick for giving me enlightment in finding my research topic. Thanks Alex for answering me a lot of questions. And thanks Kwok Tai for being my best friend in graduate school.

My final acknowledgment goes to my parents for their support and love, especially for my mother's care.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Advancement in digital audio/video and compression technologies has led to the recent wide deployment of continuous media streaming over the Internet. A wide range of applications such as video-on-demand, distance learning, and corporate telecasts and narrowcasts are now enabled by the ability to stream audio/video data from servers to clients across a wide area network. However, because of the high bandwidth requirement (e.g., a high quality video streaming application usually has a bandwidth requirement of over 1 Mb/s) and the long duration nature of digital video (e.g., from tens of minutes to several hours), server and network bandwidths are major limiting factors in achieving a *scalable* and high quality streaming service. Consequently, there has been a lot of research on developing techniques for bandwidth-efficient distribution of multimedia data to a large client population. One common solution, for example, is to use a multimedia proxy to perform some form of data caching (say, prefix caching), of popular audio/video objects, so that clients can access the cached data from their nearby proxies to reduce startup delay and conserve bandwidth.

One major problem with the multimedia proxy approach is the risk of revealing the original multimedia data to unauthorized parties. For example, when the original multimedia data are sent from a server to a multimedia proxy, anyone that eavesdrops on the communication link between the source

and the multimedia proxy can gain access to the audio/video information. Some common approaches to counter the problem are:

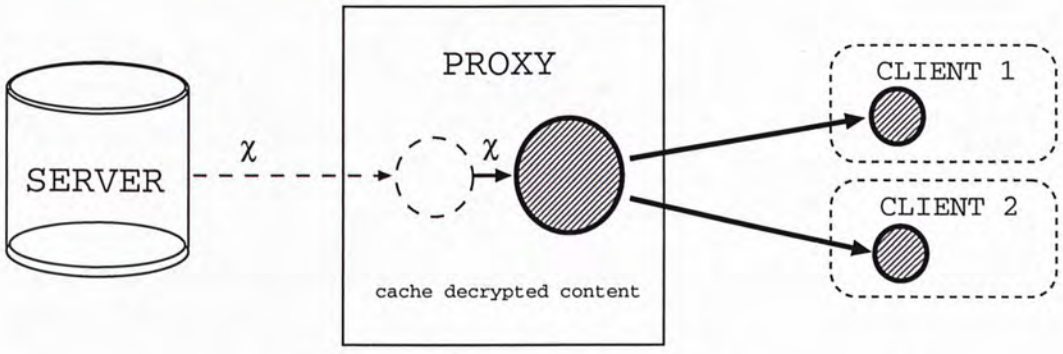


Figure 1.1: Encryption using a secret key between the server and the proxy.

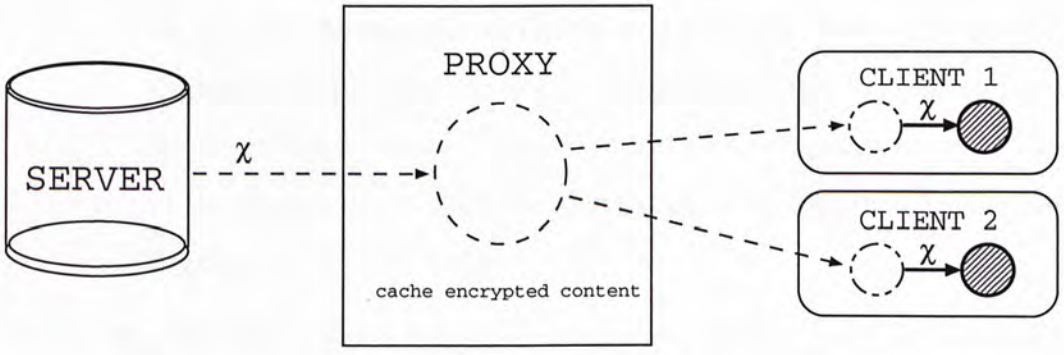


Figure 1.2: End-to-end encryption using a secret key between the server, the proxy, and the clients.

1. **Encryption using a secret key between the server and the proxy:** Figure 1.1 illustrates an approach of using a secret key between the server and the proxy for the encryption. Under this approach, the server and the proxy will exchange a secret key $\mathcal{X}$ for encrypting the audio/video data. The source encrypts the data based on the secret key $\mathcal{X}$ and sends the encrypted data to the proxy. The proxy, upon receiving the encrypted data, can perform decryption and cache the audio/video data in clear form. There are several problems with this approach, including:

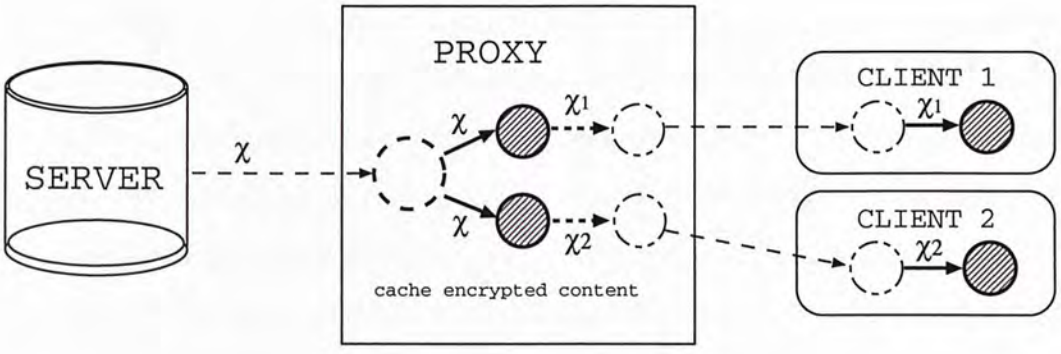


Figure 1.3: Heterogeneous secret keys between the server-proxy pair and the proxy-client pairs.

- *Data insecurity at the proxy:* Since the cached data at the proxy are the original multimedia data, any intruder who gains access to the proxy's storage can access the original data. Note that this is a major security issue since multimedia data are large in size, implying the necessity to use multiple disks. Intruders who gain access to any of the storage devices may be able to obtain the original multimedia content.
- *Data insecurity between the proxy and clients:* Since data transfer between the proxy and its clients can be over an insecure channel, one can eavesdrop on this channel and gain access to the original multimedia data.

2. **End-to-end encryption using a secret key between the server, the proxy, and the clients:** Figure 1.2 illustrates an approach of applying end-to-end encryption using a secret key between the server, the proxy, and the clients. Under this approach, the server, the proxy, and all the clients behind the multimedia proxy will share a common secret key $\mathcal{X}$. The source encrypts the multimedia data based on $\mathcal{X}$ and sends the data to the multimedia proxy. The proxy, upon receiving the encrypted data, caches the encrypted data in its local storage. Whenever

a client wants to access the multimedia data, the encrypted copy will be sent to that client. Since all clients behind the multimedia proxy know the common secret key, clients can decrypt and extract the original multimedia data. An intruder can still eavesdrop on the communication link between the proxy and the client, but it will not be able to decrypt the data. The major problem with this approach is that there is a high risk of revealing the secret key $\mathcal{X}$. The reason is that a proxy needs to support a large number of clients, and if *any* of these clients is compromised, an intruder can use the revealed secret key $\mathcal{X}$ to gain access to the original multimedia data.

3. **Heterogeneous secret keys between the server-proxy pair and the proxy-client pairs:** Figure 1.3 illustrates an approach of using heterogeneous secret keys between the server-proxy pair and the proxy-client pairs. Under this approach, the server encrypts the multimedia data based on a common secret key $\mathcal{X}$ that is shared between the server and the proxy only. Upon receiving the data, the proxy caches the encrypted data in its local storage. Whenever a client, say i, wants to access the multimedia data, the proxy will (1) decrypt the data based on the secret key $\mathcal{X}$, (2) encrypt the data based on a common secret key $\mathcal{X}_i$, which is shared between the multimedia proxy and the client i. The client i, upon receiving the encrypted data, can gain access to the original data because it knows the common secret key $\mathcal{X}_i$. The potential problem with this approach is that it requires high computational overhead at the multimedia proxy, because the proxy needs to perform decryption and encryption for every admitted client. This can be a major scalability problem which limits the number of concurrent clients that the proxy can support, and if the proxy is compromised, an intruder can obtain the secret key $\mathcal{X}$.

In this paper, we present a proxy encryption framework having the following security properties:

- The multimedia proxy will only cache the *encrypted* multimedia data and data decryption will only happen at the endpoints (e.g, the clients). Therefore, the original multimedia data will not be revealed at any intermediate node.

- The multimedia proxy will perform encryption operations only (i.e., the proxy will not perform any decryption); this reduces the computational overhead at the multimedia proxy, and hence allows one to build a more scalable proxy to support a higher number of concurrent clients. Also, even when a multimedia proxy is compromised, an intruder cannot obtain any useful information.

- Data encryption and decryption operations are based on well accepted encryption theory that it is computationally infeasible to extract the original multimedia data without knowledge of the expected decryption key.

- *Membership collusion* can be avoided, such that given (1) the decryption key of client i, (2) the encrypted data of client j, and (3) possibly all the encryption keys, the intruder still cannot derive the original multimedia data.

- The proposed approach can be extended to a multi-level proxy framework; for example, a peer-to-peer architecture.

The rest of the paper is organized as follows. In Chapter 2, we present multi-key encryption based on the *asymmetric reversible parametric sequence.* We then present a practical algorithm to implement an asymmetric reversible parametric sequence to achieve the claimed security properties for a multimedia proxy server. In Chapter 3, we present our proxy system architecture

and the proxy-server and client-proxy communication protocols. In Chapter 4, we illustrate how multi-key encryption can be extended to a multi-layer proxy architecture. Chapter 5 presents the Secure Multimedia Library (SML), a software library for developing secure multimedia applications in our architecture. In Chapter 6, we report implementation results that illustrate the achievable encryption data rate using our technique on a commodity Pentium machine, and give quantitative and qualitative analysis of the encrypted audio/video quality. Related work on multimedia proxies is presented in Chapter 7. Chapter 8 concludes.

Chapter 2

Multi-Key Encryption Theory

In this chapter, we state the theory behind the design of a multi-key secure video proxy. The main theory is based on the reversible parametric sequence (RPS) [5]. We first present the formal definition of RPS and its utilities. We then present an implementation of RPS using the multi-key RSA technique.

2.1 Reversible Parametric Sequence

Let $f : IN^2 \rightarrow IN$ be a function which has the following property: if $Y = f(X, e)$, it is *computationally infeasible* to find e given that we know X and Y.

Assume that we have a finite sequence $\{e_0, e_1, \cdots, e_N\}$ of $N+1$ elements, where the elements are not necessarily unique. Define a finite data transformation sequence $\mathcal{D} = \{D_{-1}, D_0, ..., D_N\}$ based on the function f and the finite sequence $\{e_i\}_{0 \leq i \leq N}$ such that

$$\begin{aligned} D_{-1} &= \text{original data} \\ D_i &= f(D_{i-1}, e_i) \quad \text{for } 0 \leq i \leq N. \end{aligned}$$

We have the following definitions of a reversible parametric sequence (RPS).

Definition 1 $\mathcal{D}$ is a *reversible parametric sequence* of the function f, denoted

as RPS_f, if for all $(X, Y) \in IN^2$ and $-1 \leq i < j \leq N$, there exists a computable function $\Omega_{i,j}$ such that

$$D_i = \Omega_{i,j}(D_j) \quad \text{for } -1 \leq i < j \leq N.$$

Definition 2 A RPS_f is called a *symmetric reversible parametric sequence* of f, denoted as $SRPS_f$, if the function $\Omega_{i,j}$ can be computed from the knowledge of the sub-sequence $\{e_{i+1}, \cdots, e_j\}$.

Definition 3 A RPS_f is called an *asymmetric reversible parametric sequence* of f, denoted as $ARPS_f$, if it is computationally infeasible to determine the function $\Omega_{i,j}$ based on the knowledge of the sub-sequence $\{e_{i+1}, \cdots, e_j\}$.

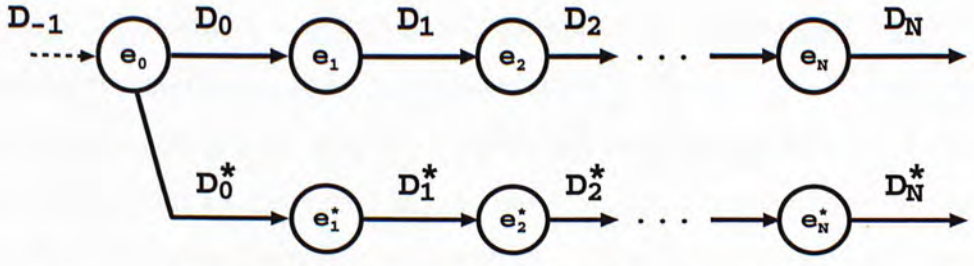


Figure 2.1: A graphical representation of two RPS_f sequences.

To understand the concepts, we use a graph to represent a reversible parametric sequence RPS_f. Figure 2.1 illustrates two RPS_f sequences. In the figure, the original data D_{-1} are transformed to D_0 using $D_0 = f(D, e_0)$. If $e_i \neq e_i^*$, then the intermediate data D_i will not be equal to D_i^*, for $1 \leq i \leq N$. For a symmetric reversible parametric sequence $SRPS_f$, one *can* compute the original data D_{-1} if given the data D_j (or D_j^*) and the sequence $\{e_0, \cdots, e_j\}$ (or $\{e_0^*, \cdots, e_j^*\}$), for $0 \leq j \leq N$. In other words, given the information $\{e_0, \ldots, e_j\}$ and D_j, one can construct a decryption function $\Omega_{-1,j}$ so as to obtain the original data D_{-1}. For an asymmetric reversible parametric sequence $ARPS_f$, one *cannot* derive the original data D_{-1} even if given the data

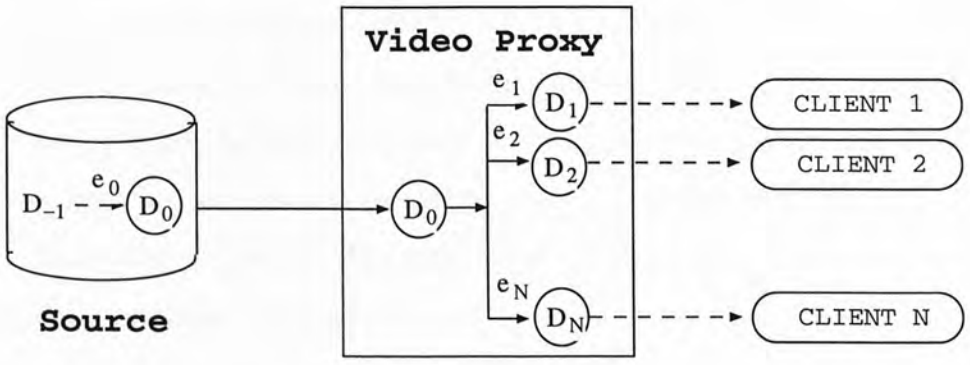


Figure 2.2: A graphical representation of an $ARPS_f$ sequence for the secure multimedia proxy considering one proxy layer.

D_j (or D_j^*) and the knowledge of the sequence $\{e_0, \cdots, e_j\}$ (or $\{e_1^*, \cdots, e_j^*\}$), for $0 \leq j \leq N$.

One can use the properties of an asymmetric reversible parametric sequence $ARPS_f$ to implement a secure multimedia proxy. To illustrate, consider a graphical representation of an $ARPS_f$ sequence in Figure 2.2. The multimedia proxy can request D_0, the encrypted version of the original data D_{-1}, from the source. Based on an encrypted key e_0, the source will transmit the encrypted data D_0 to the proxy, and the encrypted data D_0 will be cached at the proxy's local storage. When a client i requests the data, the proxy will further encrypt the encrypted data D_0 using the encryption key e_i and send the resulting encrypted data D_i to client i. Client i, upon receiving D_i, can decrypt the data to obtain the original data D_{-1}, if client i is given a decryption function $\Omega_{-1,i}$ (this is a property of reversible parametric sequences). In addition, when the encryption is carried out using an asymmetric reversible parametric sequence, then even when an entity holds on to *all* the encryption keys e_i for $0 \leq i \leq N$, it still cannot decrypt any of the encrypted data D_i being cached, for $0 \leq i \leq N$, in order to obtain the original data D_{-1}.

In general, one can use an asymmetric reversible parametric sequence $ARPS_f$ to implement a secure multimedia proxy which has the following properties:

- **Data confidentiality during transmission:** Since the original data

D_{-1} are encrypted, an intruder who eavesdrops on the communication link between the source and the multimedia proxy can only access D_0 and will not be able to extract the original data. The same property holds when an intruder eavesdrops on the link between the multimedia proxy and a client i. The intruder can only access D_i and will not be able to extract the original data.

- **End-to-end data confidentiality:** By the property of the RPS_f, decryption of the original data D_{-1} only takes place at the endpoints (e.g., the receiving clients). The multimedia proxy and clients only store the data encrypted as D_0 and D_i. Hence, even if an intruder gains access to the proxy's or client's local storage, the original data D_{-1} will not be revealed.

- **Data confidentiality against proxy intruders:** If an intruder compromises the proxy server or a client's machine, and if the encryption process is an $SRPS_f$, then the intruder, on knowing e_0 and e_i, can gain access to D_{-1}. This is because the function $\Omega_{-1,0}$ can be computed from the knowledge of e_0 and/or e_i such that the original data can be revealed by $D_{-1} = \Omega_{-1,0}(D_0)$. One example of this situation is when the encryption is carried out using a *common secret key* e_0 between the proxy and the source. In this case, access to e_0 and D_0 can reveal the original data D_{-1}. On the other hand, if the encryption process is an $ARPS_f$, then even if e_0 and e_i are compromised, the original data D_{-1} cannot be revealed, because the decryption function $\Omega_{-1,0}$ is *computationally infeasible* to determine in this case.

- **Data confidentiality against member collusion:** If the encryption process is $SRPS_f$, member collusion is possible when a client j gains access to

1. e_i and e_j (where $i, j > 0$),
2. the encrypted data D_i, and
3. the decrypting function $\Omega_{-1,j}$.

In this case, client j (i.e., the intruder) can access the original data D_{-1}. For example,

1. Given e_i, the intruder can obtain $\Omega_{0,i}$ and thus obtain $D_0 = \Omega_{0,i}(D_i)$.
2. Given the knowledge of e_j and D_0, the intruder can obtain D_j by $D_j = f(D_0, e_j)$.
3. Since the intruder knows the decryption function $\Omega_{0,j}$, the original data D_{-1} are revealed by $D_{-1} = \Omega_{-1,j}(D_j)$.

However, if the encryption process is an $ARPS_f$, the intruder cannot reveal the original data D_{-1} because the function $\Omega_{0,i}$ is computationally infeasible to determine.

2.2 Implementation of $ARPS_f$

In the previous section, we present the desirable properties of an ARPS function. However, we still need a practical and efficient algorithm to realize a secure multimedia proxy server. In our work, we use a *multi-key RSA* approach to implement an $ARPS_f$. We first present a basic overview of RSA. We then extend the concept to a multi-key RSA framework.

For standard RSA (or *single-key* RSA), one needs to generate two distinct large prime numbers p and q. Let us define

$$n = p \cdot q, \text{ and } \phi = (p-1) \cdot (q-1).$$

To encrypt a given data item D_{-1}, one has to find an encryption key e_0 such that the integer e_0 satisfies

$$1 < e_0 < \phi, \text{ and } \gcd(e_0, \phi) = 1.$$

To encrypt the data D_{-1}, we generate a cipher C based on the encryption key e_0 wherein

$$C = (D_{-1})^{e_0} \bmod n.$$

The cipher C can be transmitted over an insecure channel. The receiver needs to have a decryption key d_0 to decrypt the cipher. This decryption key d_0 is an integer and is selected based on the following criteria:

$$1 < d_0 < \phi, \text{ and } (e_0) \cdot d_0 = 1 (\bmod\ \phi).$$

Upon receiving the cipher, the receiver can decrypt the cipher C and obtain the original data D_{-1} by

$$D_{-1} = (C)^{d_0} \bmod n.$$

Let us present the extension of the single-key RSA technique to a *multi-key* RSA technique. Consider the source as an example. The source needs to generate two large prime numbers, say p and q. In addition, it needs to generate a sequence of encryption keys $\{e_i\}_{0 \le i \le N}$ such that

$$1 < e_i < \phi \qquad \text{and} \tag{2.1}$$

$$gcd(e_i, \phi) = 1 \qquad \text{for } 0 \le i \le N. \tag{2.2}$$

Moreover, one needs to generate a corresponding sequence of decryption keys d_i. The decryption key d_i has to satisfy the following two criteria:

$$1 < d_i < \phi \qquad \text{and} \tag{2.3}$$

$$(e_0 \cdot e_i) \cdot d_i = 1 \ (\bmod\ \phi). \tag{2.4}$$

Efficient computation of these decryption keys d_i can be easily achieved using the Extended Euclidean Algorithm [7]. The source will send n and the encryption keys e_i over a secure channel to the multimedia proxy, while it will encrypt the original data D_{-1} using e_0 and generate a cipher D_0 using

$$D_0 = (D_{-1})^{e_0} \bmod n. \tag{2.5}$$

The encrypted data D_0 can be sent over an insecure channel. Upon receiving the cipher D_0, the proxy can store the encrypted data in its local storage. Whenever a client i wants to access the data from the proxy, the proxy can retrieve the encrypted data D_0 from its local storage, and encrypt D_0 using the encryption key e_i by

$$D_i = (D_0)^{e_i} \bmod n. \tag{2.6}$$

The source (or the proxy) can send the decryption key d_i and n to client i over a secure channel. The encrypted data D_i, on the other hand, can be sent over an insecure channel. Client i, upon receiving the encrypted data D_i, can decrypt the data using d_i by:

$$D_{-1} = \Omega_{-1,i}(D_i) = (D_i)^{d_i} \bmod n. \tag{2.7}$$

Example: To illustrate, consider the following simple example. Suppose that the two primes are $p = 5$ and $q = 7$ (in reality, p and q will have to be large). Accordingly, $n = 5 \times 7 = 35$ and $\phi = (5-1) \times (7-1) = 24$. Let there be three encryption keys: $e_0 = 5$, $e_1 = 11$ and $e_2 = 13$. If the original data $D_{-1} = 10$, the cached data at the multimedia proxy will be

$$D_0 = (10^5) \bmod 35 = 5.$$

When client 1 requests the data, the source will generate a decryption key d_1 such that $(5 \times 11)d_1 = 1 \bmod 24$. Using the Extended Euclidean Algorithm, we have $d_1 = 7$. Therefore, the encrypted data for client 1 is

$$D_1 = (D_0)^{e_1} \bmod n = (5^{11}) \bmod 35 = 10$$

and client 1 can decrypt the data D_1 and get back the original data D_{-1} by

$$D_{-1} = (D_1)^{d_1} \bmod n = (10^7) \bmod 35 = 10.$$

When client 2 requests the data, the source will generate a decryption key d_2 such that $(5 \times 13)d_2 = 1 \bmod 24$. Using the Extended Euclidean Algorithm, we have $d_2 = 17$. Therefore, the encrypted data for client 1 is

$$D_2 = (D_0)^{e_2} \bmod n = (5^{13}) \bmod 35 = 5$$

and client 2 can decrypt the data D_2 and get back the original data D_{-1} by

$$D_{-1} = (D_2)^{d_2} \bmod n = (5^{17}) \bmod 35 = 10.$$

Theorem 4 The above proxy encryption framework is a reversible parametric sequence.

Proof: To show that the above framework is a reversible parametric sequence, we need to show that given D_i, for $i \geq 1$, we can extract D_0 and D_{-1}. Without loss of generality, let us consider D_1 as the given input. The generation of D_1 is via

$$\begin{aligned} D_1 &= (D_0)^{e_1} \bmod n. = [(D_{-1})^{e_0} \bmod n]^{e_1} \bmod n. \\ &= (D_{-1})^{e_0 \cdot e_1} \bmod n. \end{aligned}$$

Let the extracted result be R and equal to

$$\begin{aligned} R &= (D_1)^{d_1} \bmod n = [(D_{-1})^{e_0 \cdot e_1} \bmod n]^{d_1} \bmod n \\ &= (D_{-1})^{e_0 \cdot e_1 \cdot d_1} \bmod n. \end{aligned}$$

Since the encryption key e_0 and the decryption key d_1 are generated such that $e_0 \cdot e_1 \cdot d_1 = k \cdot (p-1) \cdot (q-1) + 1$ where k is an integer, we can rewrite the extracted result R as

$$R = \left[(D_{-1})(D_{-1})^{k(p-1)(q-1)}\right] \bmod n.$$

Based on Euler's theorem [7] that $X^{p-1} = 1 \bmod n$, we have the following expressions:

$$\begin{aligned} R &= (D_{-1})(D_{-1})^{k(p-1)(q-1)} = (D_{-1})1^{k(q-1)} = D_{-1} \bmod p, \\ R &= (D_{-1})(D_{-1})^{k(p-1)(q-1)} = (D_{-1})1^{k(p-1)} = D_{-1} \bmod q. \end{aligned}$$

Because p and q are primes, based on the Chinese Remainder Theorem [7], we have

$$R = (D_{-1})(D_{-1})^{k(p-1)(q-1)} = D_{-1}(\bmod n) = D_{-1}.$$

Therefore, given D_1, one can extract D_{-1} given the knowledge of d_1. We can apply a similar procedure so that given D_0, one can extract D_1 with another corresponding decryption key. In summary, given D_i, one can extract D_0 and the original data D_{-1} if the corresponding decryption key is known. ∎

Theorem 5 The above proxy encryption procedure is an asymmetric reversible parametric sequence.

Proof: To show that the above framework is an asymmetric reversible parametric sequence, all we need to show is that given the encrypted data D_i, for $i \geq 0$, it is impossible to obtain the original data D_{-1}, even if one has the knowledge of $\{e_0, e_1, e_N\}$ and n.

Since we use an RSA encryption method, finding the decryption keys in Equations (2.3)-(2.4) amounts to finding the two prime factors p and q, which is well accepted to be computationally infeasible if p and q are large and properly chosen. Therefore, it is computationally infeasible to find $\Omega_{i,j}$. ∎

Chapter 3

Multimedia Proxy: Architectures and Protocols

In this chapter, we describe in detail our server-proxy-client architecture. The multimedia server $\mathcal{S}$, the multimedia proxy $\mathcal{P}$ and various clients have been implemented in the C language on a Linux platform. Security features such as key generation, encryption and decryption are implemented using the GNU Multiply Precision (GMP 4.0) library, which provides arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. Note that GMP 4.0 provides one of the fastest possible arithmetic libraries for applications that need higher precision than what is directly supported by the basic C types.

3.1 Operations to Request and Cache Data from the Server

Let us first consider the case wherein the multimedia data are not yet cached at the proxy. Figure 3.1a illustrates the operations between the multimedia server $\mathcal{S}$ and the proxy $\mathcal{P}$, and the operations between the proxy $\mathcal{P}$ and the client i for requesting and caching the multimedia data. These operations are:

1. **Initiate connection:** The client i sends a multimedia request and its *certificate* to certify its identity to the proxy $\mathcal{P}$, and the proxy $\mathcal{P}$ forwards the request to the multimedia server $\mathcal{S}$. (Client i's certificate is $eCert_i = (u_i, \{u_i\}_{v_i})$, where u_i and v_i are i's public and private keys, respectively, for RSA cryptography, and $\{D\}_k$ denotes information D encrypted with key k.) The multimedia server keeps a record of the public key for each of its authenticated clients. To vertify the identity of the requesting client, the multimedia server $\mathcal{S}$ decrypts $\{u_i\}_{v_i}$) using u_i as the decryption key. The request will only be granted if the decrypted message equals to u_i and match the public key of i in the server's authorized list. It relies on the fact that only genuine client i has the secret key v_i, so that only client i itself can generate this $\{u_i\}_{v_i}$) which when decrypts using u_i will produce the same u_i as the copy stored by the server. If the authentication is successful, then the server $\mathcal{S}$ will proceed to the key generation operation.

2. **Key generation:** The server randomly generates two large prime numbers p and q, and computes the prime product $n = p \cdot q$, the pseudo-prime product $\phi = (n-1) \cdot (p-1)$, the encryption key e_0, re-encryption key e_i, and the corresponding decryption key d_i via Equations (2.1) and (2.2)* . The server $\mathcal{S}$ saves the parameters ϕ, e_0 and n with a unique identifier ID. It then replies back to the proxy $\mathcal{P}$ with the re-encryption key e_i, which is encrypted using the proxy's public key; and the corresponding decryption key d_i, which is encrypted using the client i's public key. Note that the proxy $\mathcal{P}$ cannot extract the decryption key d_i, but only the client i can perform the decryption to extract d_i.

*In practice, the server $\mathcal{S}$ can pre-generate a set of encryption keys $\{e_i\}$ and decryption keys $\{d_i\}$ per multimedia object per each of its authorized clients.

3. **Decryption key retrieval:** The proxy $\mathcal{P}$ replies with an acknowledgment back to the client i with the encrypted d_i. The client i decrypts using its own private key to retrieve the decryption key d_i.

4. **Data encryption and streaming:** The multimedia server $\mathcal{S}$ uses the encryption key e_0 and n to encrypt the multimedia data packets. The degree of encryption is based on the encryption configuration parameters (ECP) which we will describe in Section 3.3. The multimedia server $\mathcal{S}$ then streams the encrypted data packets to the proxy $\mathcal{P}$ via an ordinary and possibly insecure channel. Upon receiving the encrypted data packets, the proxy $\mathcal{P}$ caches the data without decryption or modification.

5. **Data re-encryption and streaming:** The proxy $\mathcal{P}$ uses the re-encryption key e_i and n to re-encrypt the already encrypted multimedia data packets in the cache. The encryption is based on the same ECP setting as what the server $\mathcal{S}$ used. The proxy $\mathcal{P}$ then streams the re-encrypted data packets to the client i via an ordinary and possibly insecure channel. Upon receiving the re-encrypted data packets, the client i can use the decryption key d_i to decrypt the received multimedia data packets.

3.2 Operations to Request Cached Data from the Multimedia Proxy

We consider the case wherein the multimedia data are already cached at the proxy. Figure 3.1b illustrates the operations between the multimedia server $\mathcal{S}$ and the proxy $\mathcal{P}$, and the operations between the proxy $\mathcal{P}$ and the client j for requesting multimedia data that are already cached by the proxy $\mathcal{P}$. These operations are:

1. **Initiate connection:** The client j sends a request to the proxy $\mathcal{P}$ with its certificate. The proxy $\mathcal{P}$ forwards the request to the server $\mathcal{S}$ with a specified unique identifier ID. The multimedia server $\mathcal{S}$ needs to authenticate that the request is indeed from the client j. If the authentication succeeds, the server $\mathcal{S}$ will go on to the key generation operation.

2. **Key generation:** The server $\mathcal{S}$ randomly generates a re-encryption key e_j and a corresponding decryption key d_j based on the ϕ, n and e_0 identified by ID. It then sends back to the proxy $\mathcal{P}$ the re-encryption key e_j, which is encrypted using the proxy's public key; and the decryption key d_j, which is encrypted using the client j's public key. Again, the proxy $\mathcal{P}$ cannot extract the decryption key d_j.

3. **Decryption key retrieval:** The proxy $\mathcal{P}$ replies back to the client j with the encrypted decryption key. The client j decrypts using its own private key to retrieve the decryption key d_j.

4. **Data re-encryption and streaming:** The proxy $\mathcal{P}$ uses the re-encryption key e_j and n to re-encrypt the cached multimedia data packets based on the previous ECP setting. It then streams the re-encrypted data packets to the client j via an ordinary and possibly insecure channel. Upon receiving the re-encrypted data packets, the client j can use the decryption key d_j to decrypt the received multimedia data packets.

3.3 Encryption Configuration Parameters (ECP)

Differ from normal text documents, multimedia objects do not require entire encryption for proper protections. For example, a MPEG-1 stream consists of three types of frames, i.e. I-frames: Interpolative, P-frames: Predictive, and B-frames: Bipredictional. I-frames are inserted for every 12 to 15 frames, and operations such as playback, forward and review can only start at I-frames.

Thus, encryption on all the I-frames would provide enough protection but only require about 0.8% encrytpion operations compared to entire encryption.

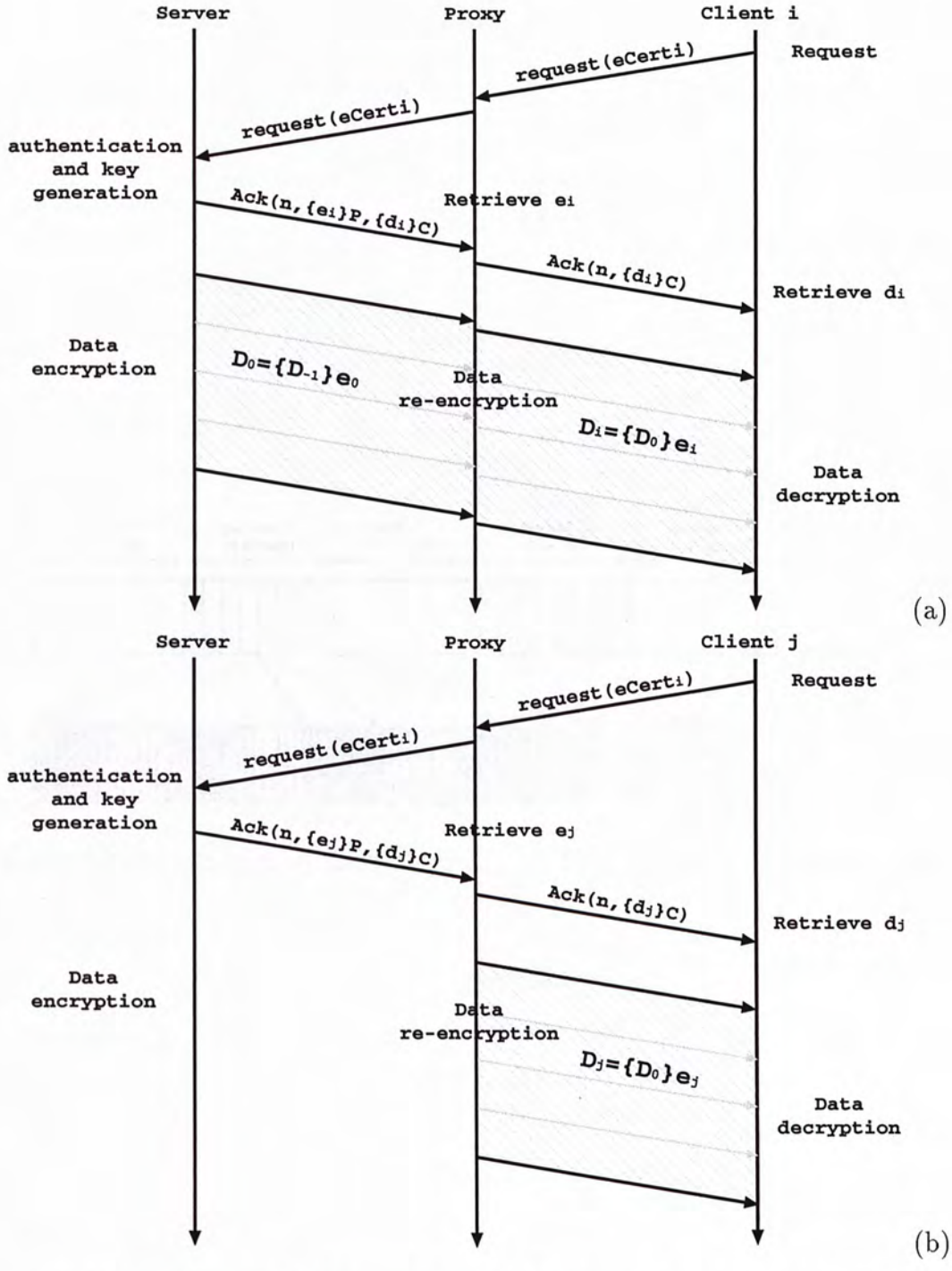
In a qualitative analysis, the degree of partial encryption affect the choppiness of the playback of the encrypted video without a proper decryption key. In applications such as video-on-demand, online-lectures, etc., system throughput or system performance is most important and encryption security could be a trade-off. In this paper, we propose to use a *general* encryption method that can reduce the encryption overhead at the server and the proxy, decryption overhead at the end-clients, for a *variety* of commonly used video encoding formats (such as MPEG-1, MPEG-2, MP3 and Quicktime). We exploit the observation that, for video encoding that accounts for inter-frame data dependencies, a video stream only needs to be encrypted up to a certain percentage for decoding to be practically useless by an unauthorized viewer, in that either the video cannot be decoded, or the quality of the decoded video will be so poor that it is unacceptable for viewing. In general, ECP specifies a packet based encryption pattern given by four adjustable parameters, namely

- S_{pkt}, the expected number of bytes in a data packet.
- E_i – the multimedia stream is to be partitioned into successive groups each having E_i consecutive packets, and a *single* packet encryption operation is to be applied to the first packet of each group.
- E_p – for the packet in which the packet encryption operation is to be applied, E_p specifies the fraction of data within the packet that should be encrypted.
- E_b – for the packet in which the encryption operation is to be applied, E_b specifies the number of encryption blocks that should be evenly distributed within that encryption packet.

In our current implementation, we use UDP as the transport protocol for

video data transmission. The entire multimedia stream will be divided into UDP packets with each packet having a payload size of $S_{pkt} = 1400$ bytes. For every $E_i \geq 1$ consecutive UDP packets, we will select the last UDP packet for encryption. For the encrypted packet, it will be further divided up into sub-blocks and only some of the sub-blocks will be encrypted. In our current implementation, the sub-block size is chosen to be 4 bytes less than the RSA key length (e.g., 60 bytes for 512-bit RSA) and the encryption will be based on this sub-block unit size. The total length of data to be encrypted within a packet is equal to $E_p \times S_{pkt}$ rounded up to the nearest multiple of the sub-block size. The encrypted sub-blocks will then be regrouped as E_b consecutive blocks of data, and the blocks will be distributed evenly across the whole packet. Once an ECP configuration is selected for a particular video object, the same configuration will be used by the server, the proxy and the end-client on their encryption or decryption operations.

Figure 3.2 illustrates a possible set of encryption configuration parameters for a multimedia streaming application, where the packet size S_{pkt} is equal to 1400 bytes, $E_i = 2$ (i.e., out of every two consecutive packets, we select the last one for encryption), the fraction E_p is equal to 0.5, and $E_b = 4$ blocks are to be evenly distributed across an encrypted packet. The four configuration parameters allow us to achieve varying degrees of encryption and levels of audio/video quality for the decoded stream. In Chapter 6, we illustrate the computational and quality tradeoffs implied by these parameters.



P is the public key of the proxy and C is the public key of the client, and $\{D\}k$ denotes information D encrypted with key k.

Figure 3.1: Operations between the source video server $\mathcal{S}$ and the proxy $\mathcal{P}$, and operations between the proxy $\mathcal{P}$ and the client i, for (a) non-cached video object and (b) cached video object.

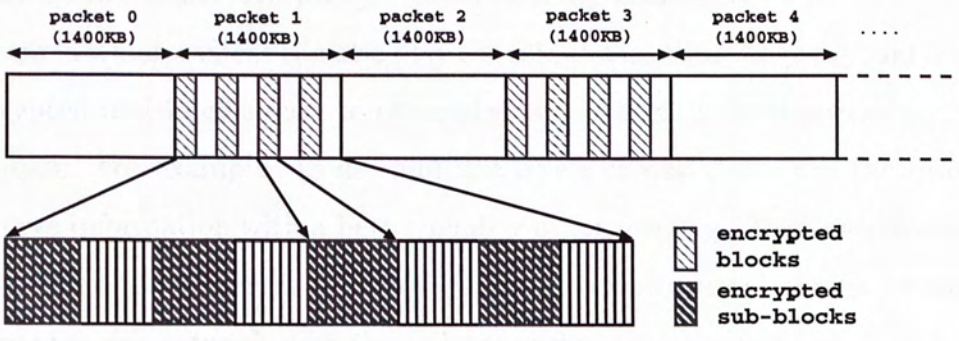


Figure 3.2: Illustration of ECP with $S_{pkt} = 1400$, $E_i = 2$, $E_p = 0.5$, and $E_b = 4$.

Chapter 4

Extension to multi-level proxy

In this chapter, we describe how the multi-key secure proxy architecture can be extended to a multi-level proxy architecture. By *multi-level proxy*, we refer to a system in which a client can also play the role of a multimedia proxy and deliver encrypted multimedia data to other clients. Such multi-level proxies are very common. For example, we use multiple layers of web proxies in the Internet to serve information with a better quality of service (e.g., lower response time in getting a file resource). A multi-level proxy architecture can also be used in a peer-to-peer network such that clients within a peer group can quickly and securely share multimedia data among themselves.

Figure 4.1 illustrates a multi-level proxy architecture. In the figure, a proxy is represented by a circle and we arrange the proxies into different layers. The source has an *encryption generator*, which is represented by a "square" near the source. For secure multimedia transmission, this encryption generator needs to:

1. Generate two large prime numbers p and q.

2. Compute $n = p \cdot q$ and $\phi = (p-1) \cdot (q-1)$.

3. Generate a set of encryption keys $\{e_i\}_{0 \leq i \leq N}$ such that $1 < e_i < \phi$ and $gcd(e_i, \phi) = 1$, where N is the number of proxies the encryption generator is willing

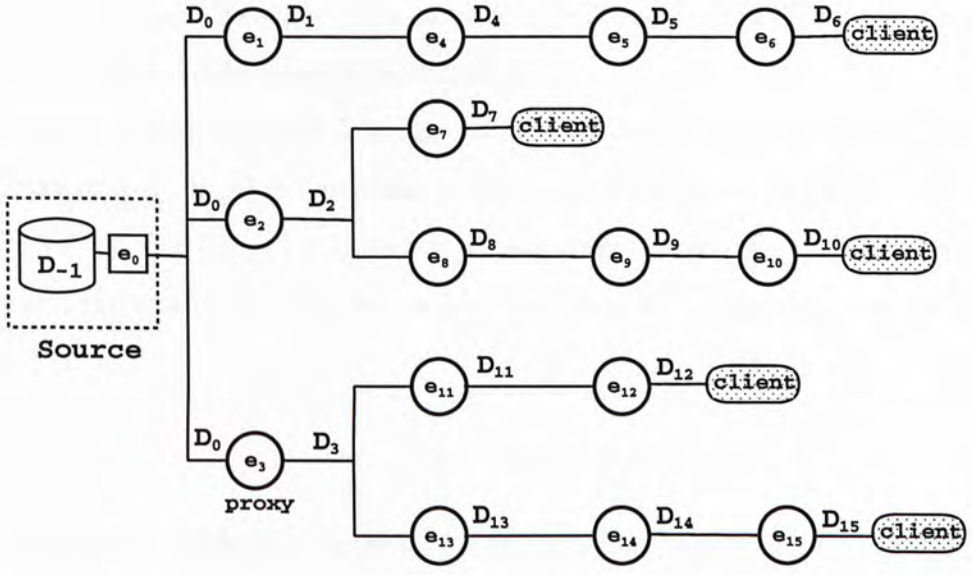


Figure 4.1: A graphical representation of a multi-level proxy architecture.

to support. Note that by selecting two very large prime numbers p and q, we can potentially generate a very large set of encryption keys.

4. Distribute each encryption key and n to each proxy on an on-demand basis. For example, the encryption generator will transmit the encryption key e_i and n to the proxy i over a secure channel.

Each proxy and the encryption generator will simply perform data input encryption. For example, to perform encryption, proxy i simply needs to perform $D_i = (D_{i-1})^{e_i} \bmod n$. To illustrate, consider the top most layer in Figure 4.1:

- The source will transmit D_0 to proxies 1, 2, and 3 where $D_0 = (D_{-1})^{e_0} \bmod n$.
- Proxy 1 will transmit D_1 to proxy 4, where $D_1 = D_0^{e_1} \bmod n = (D_{-1})^{e_0 e_1} \bmod n$.
- Proxy 4 will transmit D_4 to proxy 5, where $D_4 = D_1^{e_4} \bmod n = (D_{-1})^{e_0 e_1 e_4} \bmod n$.
- This operation will repeat until the data reach the client at the end of the chain.

This implies that all cached data along all the proxies are made secure and are encrypted potentially using different keys.

When a client of proxy k wishes to perform the data decryption, k needs to obtain the decryption key from the encryption generator and then deliver it to the client. Let $\{e_{i_k \geq 0}\}$ denote the set of encryption keys used by the proxies between the source and the proxy k. The decryption key d_k is selected such that

$$(e_{i_1} \cdot e_{i_2} \cdots e_{i_k}) \cdot d_k = 1 \,(\text{mod } \phi).$$

For example, consider the client at the top layer in Figure 4.1. The encryption generator needs to generate d_6 such that

$$(e_0 \cdot e_1 \cdot e_4 \cdot e_5 \cdot e_6) \cdot d_6 = 1 \,(\text{mod } \phi).$$

This decryption key will be encrypted using a particular end-client's public key and transmitted to proxy 6, possibly through an insecure channel, and then the proxy 6 can transmit this decryption key to that designated client within its domain.

Chapter 5

Secure Multimedia Library (SML)

We have implemented a prototype secure multimedia proxy system. The prototype is based on our Secure Multimedia Library (SML), which is a C language API that implements the secure proxy protocol described in Chapter 3. SML consists of a set of functions which can be used to develop a secure multimedia streaming system, both for the simple client-server architecture and the proxy-assisted architecture. SML is an open source project* . In this chapter, we briefly introduce the programming interface of SML. We present two scenarios and use example codes to demonstrate how to use SML API in these scenarios. A complete listing of functions in the SML library is given at the appendix.

5.1 Proxy Pre-fetches and Caches Data

Prior to any client's request, the multimedia proxy may prefetch multimedia data from the source server during the proxy's idle time. Doing so can reduce the start-up latency experienced by the clients. The proxy first initializes an SML session, and then connects to the source in anonymous mode using:

*Detailed documentation, source code, and demonstration programs can be downloaded at `http://www.cse.cuhk.edu.hk/~cslui/ANSRlab/software.html` and `http://www.cs.purdue.edu/homes/yau/sml.html`.

```
SML_SESSION proxy;
SML_InitSession(&proxy);
SML_Connect(&proxy, "source_addr", source_port);
```

Meanwhile, the source server is continually waiting for an incoming connection. An SML session between the source and the proxy is established when the source accepts the proxy connection request, as follows:

```
SML_SESSION server, *session;
SML_InitSession(&server);
session = SML_Accept(&server, port_to_bind, "client_key_list");
```

The `SML_Accept` function will generate a set of encryption and decryption keys, and associate them with the newly established session. Also, the accept function may associate a default ECP setting with the SML session, or one can install an explicit set of ECP parameters using the following function call:

```
SML_InitRps(session, Ei, Ep, Eb, Spkt, RPS_MULTI_KEY_RSA, key_length);
```

Once an SML session is set up, the source sends the chosen ECP setting to the proxy (e.g., the proxy needs the ECP to determine the network packet size for serving the cached multimedia data) through the established network connection:

```
SML_SendRps(session);
```

The ECP is received by the proxy through the function call:

```
SML_ReceiveRps(&proxy);
```

With the knowledge of the ECP by both sides, the source server may start encrypting the original multimedia data contents, denoted by D, and transmitting the encrypted contents, denoted by D', to the proxy. The size of data in each transmitted packet can be less than or equal to the specified ECP packet size – the `send` function provided by SML will perform any necessary data padding:

```
SML_TcpSendEncryptRps(session, data_packet, data_length);
```

The proxy then caches in its local storage the encrypted multimedia data packets received from the source:

```
SML_TcpReceiveRps(&proxy, data_packet, data_length);
```

5.2 Client Requests Cached Data From the Proxy

In this scenario, suppose a certified client, say i, requests the previously cached data D' from the proxy. The client i first loads its certificate $eCert_i$ (see Section 3.1) used to identify itself:

```
SML_SESSION client;
SML_InitSession(&client);
SML_LoadKeyPair(&client, "key_pair_file", "password", CRYPTO_KEY_RSA);
```

The client then connects to the proxy using the proxy's address and binding port:

```
SML_Connect(&client, "proxy_addr", proxy_port);
```

The proxy acts like the source server in accepting the connection request from the client. It then needs to use another SML session to forward the request to the actual server. That SML session must be established with the client's public key signature for proper authentication by the server:

```
SML_Signature signature;
SML_InitSignature(&signature);
session_to_client = SML_Accept(&proxy, proxy_binding_port, NULL);
SML_GetClientSignature(session_to_client, &signature);
SML_SetClientSignature(&session_to_server, signature);
SML_Connect(&session_to_server, "server_addr", server_port);
```

The proxy now holds two connections – one with the source server and the other with the client. Because the multimedia data D' are already cached by the proxy, the source does not need to perform any data encryption. Rather,

it needs to generate (i) a client-specific encryption key, e_i, to be used by the proxy to encrypt D' (the resulting data are denoted as D'') for sending to the client, and (ii) the corresponding decryption key, d_i, for the client to decrypt D'' received from the proxy to get back the *original* data D. To prevent the proxy from obtaining d_i, the source encrypts d_i with i's public key to give d_i'. The source then sends both e_i and d_i' to the proxy. In addition, it loads the ECP setting used for encryption and sends the ECP to the proxy:

```
SML_LoadRpsSetting(&server, "rps_ecp.dat", "password");
SML_SendRpsSetting(&server);
```

The proxy forwards d_i' and the ECP to the client, using:

```
RPS rps;
RPS_Init(&rps);
SML_ReceiveRpsSetting(&session_to_server);
SML_GetRpsSetting(&session_to_server,&rps);
SML_SetRpsSetting(session_to_client, rps);
SML_SendRpsSetting(session_to_client);
```

The proxy encrypts the cached D' with e_i and sends the resulting data D'' to the client:

```
SML_TcpSendReEncryptRps(&session_to_client, data_packet, data_size);
```

Finally, the client decrypts D'' using d_i to retrieve the *original* multimedia data D:

```
SML_ReceiveRpsSetting(&client);
SML_TcpReceiveDecryptRps(&client, data_packet, data_size);
```

Chapter 6

Implementation Results

In this chapter, we present our implementation results which quantify the encryption throughput, the signal-to-noise ratio (SNR) of decoded audio for an audio streaming application, and the peak signal-to-noise ratio (PSNR) and the visual quality of the decoded video for a video streaming application, in the context of our secure multimedia proxy architecture.

In our implementation, we use the ECP in Section 3.3 to control the amount of encryption applied to blocks of audio/video data. The experimental results are taken on an 800 MHz Pentium-III Linux machine with 256 MBytes of main memory. For the video experiments (Experiments 1, 2, and 3), the input data are a set of video sequences, each being an 18 MByte MPEG-1 stream or a 4.47 MByte Quicktime stream.

Experiment 1 (Encryption Throughput Analysis):
In this experiment, we consider the effect of the parameters E_p and E_i on the encryption throughput, which is denoted as ρ (in MBytes/s). Assume that we are encrypting an MPEG-1 video stream with an average bit rate of 1.5 Mb/s. Given the assumption, the average number of concurrent MPEG-1 streams that a proxy can support is M, where $M = \rho/(1.5/8)$. Table 6.1 illustrates the encryption throughput ρ and the average number of concurrent MPEG-1 streams (M) under different values of E_p and E_i, when $E_b = 1$.

As we can observe from Table 6.1, if we encrypt 25.7% of *each* video

	$\mathbf{E_p = 0.257}$		$\mathbf{E_p = 0.214}$		$\mathbf{E_p = 0.171}$	
	ρ	**M**	ρ	**M**	ρ	**M**
$\mathbf{E_i = 1}$	2.13	11.36	2.53	13.50	3.11	16.60
$\mathbf{E_i = 2}$	4.10	21.87	4.84	25.81	5.91	32.52
$\mathbf{E_i = 5}$	9.06	48.32	10.17	54.24	11.56	61.65
$\mathbf{E_i = 10}$	11.64	62.08	10.70	57.10	11.70	62.40

	$\mathbf{E_p = 0.120}$		$\mathbf{E_p = 0.086}$		$\mathbf{E_p = 0.043}$	
	ρ	**M**	ρ	**M**	ρ	**M**
$\mathbf{E_i = 1}$	4.05	21.60	5.8	30.90	10.10	53.90
$\mathbf{E_i = 2}$	7.54	40.20	10.16	54.19	11.77	62.77
$\mathbf{E_i = 5}$	11.64	62.08	11.76	62.72	11.78	62.82
$\mathbf{E_i = 10}$	11.73	62.56	11.73	62.56	11.82	63.04

Table 6.1: Effect of E_p and E_i on the encryption throughput ρ (in unit of MBytes/s) and the average number of MPEG-1 streams M when $E_b = 1$.

packet (i.e., $E_i = 1$), the encryption throughput achieved is only around 2.13 MBytes/s, which implies that we can only concurrently handle about 11 MPEG-1 streams. On the other hand, if we encrypt one video packet for every 10 packets (i.e., $E_i = 10$) and for each video packet encrypted, we encrypt only 4.3% of its data (i.e., $Ep = 0.043$), then the encryption throughput improves to 11.82 MBytes/s, which implies that we can concurrently support about 63 MPEG-1 streams. In general, the smaller the value of E_p and the higher the value of E_i, the higher the achieved encryption throughput, and the higher the number of concurrent video streams that can be supported.

Table 6.2 illustrates the effect of E_i and E_b under two different encryption percentage parameters E_p. As we can observe, the parameter E_b has little effect on the encryption throughput.

Experiment 2 (Peak Signal-to-Noise Analysis):

In this experiment, we consider the effect on the video quality as we vary the parameters E_i, E_p, and E_b. One way to quantitatively evaluate the video quality is by the peak signal-to-noise ratio. In general, for a frame size of $m \times n$

	encryption throughput ρ (MB/sec)			
	$\mathbf{E_i = 1}$	$\mathbf{E_i = 2}$	$\mathbf{E_i = 5}$	$\mathbf{E_i = 10}$
$\mathbf{E_b = 1}$	2.13	4.10	9.06	11.64
$\mathbf{E_b = 2}$	2.12	4.09	9.01	11.66
$\mathbf{E_b = 3}$	2.12	4.09	9.07	11.65

(a) $E_p = 0.257$

	encryption throughput ρ (MB/sec)			
	$\mathbf{E_i = 1}$	$\mathbf{E_i = 2}$	$\mathbf{E_i = 5}$	$\mathbf{E_i = 10}$
$\mathbf{E_b = 1}$	3.11	5.91	11.56	11.70
$\mathbf{E_b = 2}$	3.11	5.89	11.67	11.72
$\mathbf{E_b = 4}$	3.11	5.89	11.60	11.72

(b) $E_p = 0.171$

Table 6.2: Effect of E_i and E_b on the encryption throughput ρ (in MBytes/s) for (a) $E_p = 0.257$ and (b) $E_p = 0.171$.

with a total of l frames and 3 color channels (i.e., red, green, and blue, each represented by a 8-bit number), the peak signal-to-noise ratio (SNR_{peak}) is calculated using the following equation:

$$SNR_{peak} = 10 \times \log_{10} \frac{255^2}{\left(\frac{\sum_{x=1}^{m} \sum_{y=1}^{n} \sum_{z=1}^{l} \sum_{c=1}^{3} (P_1(x,y,z,c) - P_2(x,y,z,c))^2}{3mnl} \right)}$$

where $P_1(x, y, z, c)$ means that the pixel value at coordinates (x, y) in the z-th frame for color channel c, where $c = 1$, $c = 2$, and $c = 3$ correspond to the color channels red, green, and blue, respectively. In our experiment, the values of m,n, and l are 640, 480, and 1000, respectively. Values of P_1 are obtained from the video frames decoded by a client which does not have access to the decryption key, while values of P_2 are obtained from the original video frames. Note that a lower value of SNR_{peak} indicates that the encrypted stream is more distorted from the original video stream.

	peak signal-to-noise ratio $\mathbf{SNR_{peak}}$		
	$\mathbf{E_p = 0.257}$	$\mathbf{E_p = 0.214}$	$\mathbf{E_p = 0.171}$
$\mathbf{E_i = 1}$	7.83	8.01	8.52
$\mathbf{E_i = 2}$	9.13	8.30	8.70
$\mathbf{E_i = 5}$	11.17	9.81	10.73
$\mathbf{E_i = 10}$	13.06	11.26	12.87

	peak signal-to-noise ratio $\mathbf{SNR_{peak}}$		
	$\mathbf{E_p = 0.120}$	$\mathbf{E_p = 0.086}$	$\mathbf{E_p = 0.043}$
$\mathbf{E_i = 1}$	9.32	9.39	8.85
$\mathbf{E_i = 2}$	9.48	9.87	9.51
$\mathbf{E_i = 5}$	10.81	11.39	11.33
$\mathbf{E_i = 10}$	12.60	13.26	12.82

Table 6.3: Effect of E_p and E_i on the peak signal-to-noise ratio SNR_{peak} on MPEG-1 video when $E_b = 1$.

Table 6.3 and Table 6.4 illustrate the peak signal-to-noise ratio SNR_{peak} for different values of E_p and E_i with $E_b = 1$ for MPEG-1 and Quicktime video, respectively. Note that even when we encrypt one out of 10 video packets, and for a selected packet, we only encrypt 4.3% of the data, we can still obtain a very low value of SNR_{peak}. This experiment indicates that (1) we can apply this encryption technique for different video formats (e.g., MPEG1 or Quicktime) and, (2) we only need to encrypt a small fraction of the video data to achieve *both* high encryption throughput and high video distortion.

Experiment 3 (Comparison of visual quality of encrypted video): In this experiment, we consider the effect of the ECP parameters E_i, E_p and E_b on the *visual quality* of the video. Figure 6.1 illustrates the quality of five consecutive MPEG-1 video frames* . Figure 6.1(a) is the original video frames that a client can decode given access to the decryption key. Figures 6.1(b)-(e) are the corresponding five video frames when decoded without the decryption

*The original and encrypted video/audio clips used in the experiments can be viewed or downloaded at `http://www.cse.cuhk.edu.hk/~cslui/ANSRlab/software/sml`

	peak signal-to-noise ratio $\mathbf{SNR_{peak}}$		
	$\mathbf{E_p = 0.257}$	$\mathbf{E_p = 0.214}$	$\mathbf{E_p = 0.171}$
$\mathbf{E_i = 1}$	11.38	11.56	11.72
$\mathbf{E_i = 2}$	12.15	12.13	12.27
$\mathbf{E_i = 5}$	12.63	12.48	12.57
$\mathbf{E_i = 10}$	12.97	12.76	12.84

	peak signal-to-noise ratio $\mathbf{SNR_{peak}}$		
	$\mathbf{E_p = 0.120}$	$\mathbf{E_p = 0.086}$	$\mathbf{E_p = 0.043}$
$\mathbf{E_i = 1}$	12.13	12.28	12.48
$\mathbf{E_i = 2}$	12.55	12.48	12.77
$\mathbf{E_i = 5}$	12.97	12.84	12.98
$\mathbf{E_i = 10}$	13.24	12.98	13.09

Table 6.4: Effect of E_p and E_i on the peak signal-to-noise ratio SNR_{peak} on Quicktime video when $E_b = 1$.

key. Note that the video quality is the worst when the ECP parameters are $E_i = 1$ and $E_p = 0.043$, which corresponds to encrypting 4.3% of the data for every video packet. Note that when we select $E_i = 10$, $E_p = 0.043$, and $E_b = 1$ (this corresponds to Figure 6.1(e)), the visual quality of the video is still unacceptable for viewing. This shows that we can achieve high encryption throughput (i.e., around 11.82 MBytes/s or about 63 concurrent MPEG-1 streams from Table 6.1) and, at the same time, ensure that those clients which do not possess the decryption keys will get unacceptable video quality on viewing. Figure 6.2 shows the corresponding results for five consecutive Quicktime video frames. Similar conclusions can be drawn from the Quicktime results.

Experiment 4 (Discarding encrypted data analysis):

In this experiment, we consider the effect on the video quality when an authorized party just try to discard all of the encrypted data before decoding an encrypted stream without having the proper decryption key. We consider two different ways to discard those encrypted data. The first one is to drop all of the encrypted data, and the second one is to fill all of the encrypted data with

	encryption throughput ρ (MB/sec)		
	direct	**drop**	**fillzero**
$\mathbf{E_i = 1}$	8.85	8.26	8.26
$\mathbf{E_i = 2}$	9.51	8.70	8.60
$\mathbf{E_i = 5}$	11.33	9.45	10.10
$\mathbf{E_i = 10}$	12.82	10.54	11.10

(a) MPEG-1 streams

	encryption throughput ρ (MB/sec)		
	direct	**drop**	**fillzero**
$\mathbf{E_i = 1}$	12.48	13.08	12.46
$\mathbf{E_i = 2}$	12.77	13.11	12.76
$\mathbf{E_i = 5}$	12.98	13.05	12.97
$\mathbf{E_i = 10}$	13.09	13.07	13.09

(b) QuickTime streams

(**direct**: decode directly; **drop**: drop encrypted data; **fillzero**: fill encrypted data with zeros.)

Table 6.5: Effect of discarding encrypted data on the peak signal-to-noise ratio SNR_{peak} when $E_b = 1$ and $E_p = 0.043$.

zeros.

Table 6.5 illustrates the peak signal-to-noise ratio SNR_{peak} for dropping encrypted data and filling encrypted data with zeros under four different ECP encryption schemes. Note that we get similar, or even lower values of $SNRpeak$ when discarding encrypted data, compared to direct decoding of the encrypted streams. Figure 6.1(e-g) and 6.2(e-g) show the five video frames decoded in each streams respectively, they suggested that discarding encrypted data does not help in improving the visual quality. This experiment indicates that an unauthorized party cannot get a better decoding quality by means of discarding the encrypted video data.

Experiment 5 (Signal-to-Noise Analysis for Audio Streaming Application):

In this experiment, we consider the effect on the audio quality as we vary the parameters E_i, E_p and E_b. The audio clip used in this experiment is a MPEG-1 layer 3 encoded audio file at bit rate 128 kb/s. We compute the signal-to-noise ratio (SNR) with the Matlab program using the following equation:

$$SNR = \frac{\sum_{i=1}^{n} original(i)^2}{\sum_{i=1}^{n} (original(i) - cipher(i))^2}$$

where $original(i)$ denotes the i-th sample in the wave form decoded from the original audio stream, and $cipher(i)$ denotes the i-th sample in the wave form decoded from the encrypted audio stream without the decryption key. In this experiment, n equals 44100, which means that samples from the first one second of the audio stream are used. Note that a lower value of SNR indicates that the encrypted audio stream is acoustically more distorted from the original audio stream, while an SNR value of *infinity* indicates that the measured samples are exactly identical to those in the original audio stream.

Table 6.6 illustrates the signal-to-noise ratio SNR for different values of E_i and E_p, when $E_b = 1$. Again, we observe that one does not need to encrypt all the audio packets to sufficiently distort the audio signal. In general, our proposed ECP method allows one to simultaneously achieve high encryption throughput and low audio fidelity during unauthorized access.

	signal-to-noise ratio SNR				
	$\mathbf{E_p = 0.214}$	$\mathbf{E_p = 0.171}$	$\mathbf{E_p = 0.120}$	$\mathbf{E_p = 0.086}$	$\mathbf{E_p = 0.043}$
$\mathbf{E_i = 1}$	0.9104	0.7720	0.8571	0.8429	0.8264
$\mathbf{E_i = 2}$	0.5831	0.5608	0.5614	0.5585	0.5707
$\mathbf{E_i = 5}$	0.5479	1.0334	1.0360	13.6172	2.3095
$\mathbf{E_i = 10}$	1.0494	1.0494	1.0494	25.1848	25.1849

Table 6.6: Effect of E_p and E_i on the signal-to-noise ratio SNR on MP3 audio when $E_b = 1$.

(a) Original frames

(b) Encrypted frames with $E_i = 10$, $E_p = 0.043$ and $E_b = 1$.

(c) Encrypted frames with $E_i = 5$, $E_p = 0.043$ and $E_b = 1$.

(d) Encrypted frames with $E_i = 2$, $E_p = 0.043$ and $E_b = 1$.

(e) Encrypted frames with $E_i = 1$, $E_p = 0.043$ and $E_b = 1$.

(f) Encrypted frames with $E_i = 1$, $E_p = 0.043$ and $E_b = 1$. (encrypted data are filled with zeros.)

(g) Encrypted frames with $E_i = 1$, $E_p = 0.043$ and $E_b = 1$. (encrypted data are being dropped.)

Figure 6.1: Quality of five consecutive MPEG-1 video frames under different ECP parameters.

(a) Original frames

(b) Encrypted frames with $E_i = 10$, $E_p = 0.043$ and $E_b = 1$.

(c) Encrypted frames with $E_i = 5$, $E_p = 0.043$ and $E_b = 1$.

(d) Encrypted frames with $E_i = 2$, $E_p = 0.043$ and $E_b = 1$.

(e) Encrypted frames with $E_i = 1$, $E_p = 0.043$ and $E_b = 1$.

(f) Encrypted frames with $E_i = 1$, $E_p = 0.043$ and $E_b = 1$. (encrypted data are filled with zeros.)

(g) Encrypted frames with $E_i = 1$, $E_p = 0.043$ and $E_b = 1$. (encrypted data are being dropped.)

Figure 6.2: Quality of five consecutive Quicktime video frames under different ECP parameters.

Chapter 7

Related Work

Recent research on video proxies has mainly focused on caching strategies and replacement algorithms. Sen and Towsley [8] present how *prefix caching* at a proxy can help to shield clients from large start-up delay, low throughput, and high packet loss. Guo *et al.* [3] propose the use of a prefix-caching proxy in conjunction with a periodic broadcasting technique to improve system scalability. Focusing on implementation and protocol issues, Cruber *et al.* [2] show how to realize proxy prefix caching by using the Real-Time Streaming Protocol (RTSP). Rejaie *et al.* [6] present a fine-grained replacement algorithm for a multimedia proxy, which targets layered-encoded streams. Kangasharju *et al.* [4] present a caching model of layered-encoded multimedia streams, and propose utility heuristics whose performance are evaluated through their caching model.

There are only a small number of papers emphasizing security issues in a video proxy. Griwodz *et al.* [1] propose an approach in which the proxy stores the major part of the video streams which are intentionally corrupted. The proxy can distribute the corrupted part via multicast transmission, while the origin server will supply the part for data reconstruction in a unicast manner. Since the original server must perform data encryption for each client, this is not a scalable solution. Tosun and Feng [10] propose a much more scalable approach based on a lightweight encryption algorithm for multimedia streams.

When a client makes a request, the proxy will decrypt the locally stored encrypted data and encrypt it again using the client's encryption key. The major drawback with their approach is that the use of light-weight encryption offers no proven resilience against attacks on data confidentiality. Furthermore, the need for decryption operations at the proxy results in higher computational overhead. Shi and Bhargava [9] present an MPEG video encryption algorithm called VEA such that one can encrypt a video stream multiple times (each with, say, a client-specific key) and still decrypt the video in a single operation using a composite decryption key. However, VEA is not resilient against plaintext attack. Hence, while the approach is highly efficient, more determined adversaries can obtain the VEA secret key with feasible efforts.

Chapter 8

Conclusion

We have presented the design and implementation of a multi-key secure multimedia proxy architecture. Our design is based on the notion of an *asymmetric reversible parametric sequence* (ARPS). We discussed how ARPS can be applied to a general client-proxy-server architecture. To practically achieve the confidentiality properties of ARPS, we presented a multi-key RSA technique, and proved that the technique realizes an ARPS. In summary, our theoretical results show that the proposed architecture can achieve comprehensive data confidentiality that is *provably resilient* against attacks, given standard computability assumptions. We have an implementation of our multimedia streaming architecture – consisting of server, proxy and client – on commodity Pentium III/800 MHz machines running Linux. Our implementation results empirically demonstrate how a set of four ECP parameters can trade off encryption throughput against amount of data to protect, for a number of standard MPEG-1 and Quicktime video sequences, and a number of MP3 audio sequences. Our results indicate that it is possible to *simultaneously* achieve high encryption throughput and extremely low audio/video quality (in terms of decoded audio SNR and both PSNR and the visual quality of decoded video frames) during unauthorized access. For example, by using $E_i = 10$ and $E_p = 0.043$, a single Pentium III/800 MHz machine can concurrently

sustain more than 64 *distinct* MPEG-1 video streams, while giving good protection for the original video data. We also presented our Secure Multimedia Library (SML) that provides a comprehensive API for building secure multimedia streaming applications based on the asymmetric parametric sequence. We believe that the proposed system offers an effective approach for delivering multimedia contents in a secure manner.

Appendix A

Function Prototypes of Secure Multimedia Library (SML)

A.1 CONNECTION AND AUTHENTICATION

A.1.1 Create SML Session

SML uses a SML_SESSION to identify each unique connection. For example, when a proxy is serving two concurrent end clients while accessing the source server, there will be three active SML_SESSION's in the proxy where each SML_SESSION has its own underlying transport layer sockets, encryption parameters, and connection states. To create a new SML_SESSION, a proxy needs to declare and call the initialization function:

```
SML_SESSION session;
SML_InitSession(SML_SESSION* session);
```

A.1.2 Public Key Manipulation

SML uses public key cryptography to perform client authentication and session key encryption. SML has functions for key generation, key saving, and key retrieval. These function calls enable the creation of new key pair with a certain bit length, the storing of a key pair in a file with password protection,

and the loading of a key pair from the file. Currently, SML only supports the RSA method (however, hooks are provided to extend it to other methods). One can use the constant CRYPTO_KEY_RSA to specify the key type to be RSA. The key creation and key saving/loading functions are:

```
SML_CreateKeyPair(SML_SESSION* session,
    int bit_length, int public_exponent, int key_type);
SML_SaveKeyPair(SML_SESSION* session,
    char* key_filename, char* password);
SML_LoadKeyPair(SML_SESSION* session,
    char* key_filename, char* password, int key_type);
```

There are also helper functions that enable key manipulation without an SML_SESSION. In fact, the above functions invoke the following lower level functions for doing the real work. To use these lower level functions, one needs to declare and initialize a CRYPTO_KEY_PAIR variable. The helper function prototypes are:

```
CRYPTO_InitKeyPair(CRYPTO_KEY_PAIR* pair);
CRYPTO_CreateKeyPair(CRYPTO_KEY_PAIR* pair,
    int bit_length, int public_enponent, int key_type);
CRYPTO_SaveKeyPair(CRYPTO_KEY_PAIR pair,
    char* key_filename, char* password);
CRYPTO_LoadKeyPair(CRYPTO_KEY_PAIR* pair,
    char* key_filename, char* password, int key_type);
```

A.1.3 Authentication

SML has an authentication protocol integrated into its connect and accept functions, which gives the server the ability to verity the identity of a requesting client. The authentication is based on the correctness of the client's public key signature and the identification of the client's public key compared to that saved by the server. The server keeps a list of the public keys of all its approved clients, and the public key list file must be specified in the accept function call which will be described in the next section. SML provides helper functions for manipulating the public key list file, includomg adding an entry to, removing

an entry from, and searching for an entry in the list file. Each entry in the list file consists of a user name and the user's public key in plaintext. These helper functions are:

```
CRYPTO_AddKeyToListFile(CRYPTO_KEY key,
    char* user_name, char* key_list_filename);
CRYPTO_RemoveKeyFromListFile(CRYPTO_KEY key,
    char* key_list_filename);
CRYPTO_RemoveUserFromListFile(char* user_name,
    char* key_list_filename);
BOOL CRYPTO_QueryUserFromListFile(char* user_name,
    int user_name_buf_size, CRYPTO_KEY key, char* key_list_filename);
```

The first function will create a new file if the file specified by the filename does not exist. The second function and third function allow to delete an entry either by specifying the user name or the user's public key. The last function will search for the specified user's public key in the list file. It returns TRUE and the corresponding user name if the user exists, and returns FALSE otherwise.

A.1.4 Connect and Accept

A client connects to a server by calling the connect function with the server host address and the server's listening port. A server waits on a port for incoming clients by calling the accept function. The accept function blocks until a connection is established or when an error occurs. Notice that the accept function, on success, returns a pointer to a new SML_SESSION which corresponds to the newly established connection, while the original SML_SESSION can be used to wait for other incoming client requests.

```
SML_SESSION* SML_Accept(SML_SESSION* session,
    int port_to_listen, char* key_list_filename);
BOOL SML_Connect(SML_SESSION* session,
    char* server_host_name, int server_listening_port);
```

A.1.5 Close Connection

Once an SML_SESSION is finished or when one wishes to shut down a connection immediately, the destroy function can be called to release any allocated resources and close the underlying transport layer socket.

```
SML_DestroySession(SML_SESSION* session);
```

A.2 SECURE DATA TRANSMISSION

A.2.1 Asymmetric Reversible Parametric Sequence and Encryption Configuration Parameters

SML uses the Asymmetric Reversible Parametric Sequence (ARPS) for the encryption and decryption of bulk data. To reduce complexity, SML uses encryption configuration parameters (ECP) to control the extent of encryption and decryption on the bulk data. ECP values and ARPS parameters can be configured at the server side. The client side must then receive and use the server-selected setting in order to perform proper decryption. These related functions are:

```
SML_InitRpsSetting(SML_SESSION* session,
    int EI, int EP, int EB, int pkt_size,
    RPS_ALGORITHM, int rps_key bit_length);
SML_SendRpsSetting(SML_SESSION* session);
SML_ReceiveRpsSetting(SML_SESSION* session);
```

The first function is used by a server to configure the ECP setting and generate a pair of ARPS encryption and decryption keys for the given SML_SESSION. The second to last parameter specifies the ARPS algorithm to be used. It must be MULTI_KEY_RSA for the current version. The last parameter specifies the bit length of the ARPS keys to be generated. In order for the client to achieve proper data decryption, the client must have the same ECP setting and the

corresponding ARPS decryption key. One can use the second and third functions to send the ECP setting and decryption key to the client. All the secret information sent will be encrypted using the client's public key.

A.2.2 Bulk Data Encryption and Decryption

SML integrates data encryption and decryption with network socket functions. Thus, only a single function call is required to encrypt data and send the data though the network, or to receive data from the network and decrypt the data. SML provides *both* TCP and UDP versions of the functions. Notice, however, that SML will neither handle packet loss nor out-of-order packets when the UDP functions are used. The reason why we do not provide packet loss recovery and in-order packet delivery is that different multimedia streaming applications may have different delivery requirements. Therefore, we allow users to adapt SML use to their own application needs. To encrypt or decrypt data using the configured ECP setting and APRS keys, one simply allocates the required memory buffers and calls the following functions:

```
SML_TcpSendEncryptRps(SML_SESSION* session,
    char* send_buffer, int buffer_size);
SML_UdpSendEncryptRps(SML_SESSION* session,
    char* send_buffer, int buffer_size);
SML_TcpReceiveDecryptRps(SML_SESSION* session,
    char* receive_buffer, int buffer_size);
SML_UdpReceiveDecryptRps(SML_SESSION* session,
    char* receive_buffer, int buffer_size);
```

Note that packets of size smaller than or equal to the packet size specified in the function SML_InitRps() can be used. (However, packets of larger size are not allowed.) SML will automatically perform packet padding when necessary.

A.2.3 Entire Data Encryption and Decryption

For certain information other than entertainment audio/video, full data encryption may be preferred for security reasons. SML also provides function calls to encrypt and decrypt all data, without using ECP. Both TCP and UDP are supported. Since total data encryption or decryption does not depend on the ECP setting, the packet size will not be limited by that specified in the function SML_InitRps() in this case. The related functions are:

```
SML_TcpSendEncrypt(SML_SESSION* session,
    char* send_buffer, int buffer_size);
SML_UdpSendEncrypt(SML_SESSION* session,
    char* send_buffer, int buffer_size);
SML_TcpReceiveDecrypt(SML_SESSION* session,
    char* receive_buffer, int buffer_size);
SML_UdpReceiveDecrypt(SML_SESSION* session,
    char* buffer, int buffer_size);
```

A.3 Secure Proxy Architecture

A.3.1 Proxy-Server Connection

As described previously, a server needs to identify public keys from approved clients – these keys will be used to encrypt the decryption key required by a client, which should be kept secret even from the proxy. Therefore, the proxy should forward the client's public key (received from the client side SML_SESSION) to the server (sent through the server side SML_SESSION before calling the connect function). These related functions are:

```
SML_GetClientPublicKey(SML_SESSION *session, CRYPTO_KEY *key);
SML_SetClientPublicKey(SML_SESSION *session, CRYPTO_KEY key);
```

A.3.2 ARPS and ECP

SML performs ARPS cryptography. In order for the cryptography to work correctly, both ECP and the ARPS parameters must be preserved during a

chain of encryption, re-encryption and decryption. For example, the first time a proxy requests a particular multimedia object from a server, the server can select its own ECP setting and randomly generate the ARPS parameters for the SML_SESSION. However, when another client requests that cached object from the proxy, the proxy must request from the server a new re-encryption and decryption key pair based on the previously chosen ARPS parameters, and the proxy must use the previously chosen ECP setting.

The server can use the following functions to store and retrieve RPS settings. However, it is not necessary to deal with re-encryption key generation since both the functions
SML_InitRpsSetting() and SML_LoadRpsSetting() will internally generate a random re-encryption key pair, and the decryption keys will be encrypted using the authorized client's public key.

```
SML_SaveRpsSetting(SML_SESSION* session, char* filename);
SML_LoadRpsSetting(SML_SESSION* session, char* filename);
```

Similar to the public key issue, a multimedia proxy is required to retrieve the ARPS setting from the server side's SML_SESSION and then use the same setting in the client side's SML_SESSION. One can use the following `get` and `set` functions to achieve this requirement.

```
SML_GetRpsSetting(SML_SESSION *session, RPS_SETTING *rps);
SML_SetRpsSetting(SML_SESSION *session, RPS_SETTING rps);
```

A.3.3 Initial Sever Encryption

If a client requests a multimedia object which is not cached by the proxy, the proxy needs to request the object from the server. The server simply encrypts the data packets using the functions described in Session A.2.2 as in the client-server scenario. However, instead of decrypting the data packets, the proxy caches them without any decryption and modification. Note that the proxy does not have the capability to decrypt the data packets since the decryption

key is encrypted using the client's public key. The following SML functions will receive the data packets without performing any decryption operations.

```
int SML_TcpReceiveRps(SML_SESSION* session,
    char* packet_buffer, int buffer_size);
int SML_UdpReceiveRps(SML_SESSION* session,
    char* packet_buffer, int buffer_size);
```

Since the size of data packets may change after ARPS encryption, the above two functions will return the actual size of the data received. And since the packet size may increase after encryption, one may need to allocate a buffer of a size larger than the plaintext data packet size. The following function allows to determinate the maximum packet size required:

```
int SML_GetMaxRpsPacketSize(SML_SESSION* session);
```

A.3.4 Proxy Re-Encryption

A proxy will receive the APRS re-encryption key along with other settings during the SML_ReceiveRpsSetting() function call. The proxy can use the following functions to re-encrypt those cached data packets.

```
SML_TcpSendReEncryptRps(SML_SESSION* session,
    char* packet_buffer, int packet_size);
SML_UdpSendReEncryptRps(SML_SESSION* session,
    char* packet_buffer, int packet_size);
```

A.3.5 Client Decryption

A client will receive the ARPS decryption key along with other settings during the SML_ReceiveRpsSetting() function call. The client can simply use the same decryption functions described in Session A.2.2 to retrieve the plaintext data packets.

Bibliography

[1] Carsten Griwodz, Oliver Merkel, Jana Dittmann, and Ralf Steinmetz. Protecting vod the easier way. In *Proceeding of the 6th ACM International Multimedia Conference*, pages 21–28, September 1998.

[2] Stephane Gruber, Jennifer Rexford, and Andrea Basso. Protocol considerations for a prefix-caching proxy for multimedia streams. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[3] Yang Guo, Subhabrata Sen, and Don Towsley. Prefix caching assisted periodic broadcast: Framework and techniques to support streaming for popular videos. In *IEEE ICC*, 2002.

[4] J. Kangasharju, F. Hartanto, M. Reisslein, and K. W. Ross. Distributing layered encoded video through caches. In *Proceedings of IEEE Infocom 2001*, pages 1791–1800, Anchorage, Alaska, April 2001.

[5] Refik Molva and Alain Pannetrat. Scalable multicast security in dynamic groups. In *Proceeding of the 6th ACM Conference on Computer and Communications Security*, pages 101–111, November 1999.

[6] Reza Rejaie, Mark Handley, Haobo Yu, and Deborah Estrin. Proxy caching mechanism for multimedia playback streams in the internet. In *Proceedings of the 4th International Web Caching Workshop*, San Diego, CA., March 1999.

[7] Bruce Schneier. *Applied Cryptography.* John Wiley and Sons, New York, 1996.

[8] Subhabrata Sen and Don Towsley. Proxy prefix caching for multimedia streams. In *IEEE INFOCOM, New York*, March 1999.

[9] Changgui Shi and Bharat Bhargava. A fast mpeg video encryption algorithm. In *Proceeding of the 6th ACM International Multimedia Conference*, pages 81–88, September 1998.

[10] Ali Saman Tosun and Wu chi Feng. Secure video transmission using proxies. In *Technical Report, Computer and Information Science, Ohio State Univeristy*, 2002.