MASSACHUSETTS INSTITUTE
OCT 29 1997
WITHDRAWN FROM MIT LIBRARIES

Incremental Cryptography

by

Yoav Yerushalmi

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

© Yoav Yerushalmi, MCMXCVII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by ....
Shafi Goldwasser
Professor of Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Incremental Cryptography

by

Yoav Yerushalmi

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Computer Science and Engineering

Abstract

Given a cryptographic operation and data to be operated upon, standard schemes compute the cryptographic operation from scratch. Incremental schemes instead attempt to use a previously computed result if available on a previous version of the document, along with a list of changes performed to obtain the later document. The goal is to yield a faster computation.

This technique can be applied to a wide variety of situations. Examples include creating dynamically updated MACs, digital signatures, or encryptions. Ideally, the apparent speed improvements will make integrating this feature into a system as a default viable. Users will not notice the delays inherent in traditional cryptography, and so will accept it.

The problem of efficiency of computation of ciphers will be analyzed in terms of perceived speed. A solution for the long delays after editing a document and waiting for a cryptographic operation to complete will be proposed. Finally, a testbed implementation of both a MAC generator and a public-key encryptor for emacs is written and analyzed.

Thesis Supervisor: Shafi Goldwasser
Title: Professor of Computer Science

Acknowledgments

I would like to acknowledge first and foremost, my advisor, Shafi Goldwasser, whose help and encouragement got me this far, and hopefully further. She was patient with me even when I did ask some really clueless questions.

Next I would like to thank my extended family for their support and encouragement throughout my educational career. Thanks Mom and Dad, Ella, Freda, Izak, Yael, Keren, and Liat.

Of course, I couldn't have done this without the support of all my friends, of which there are far too many to list here, but you know who you are. Special thanks to Chad who was always there to offer support when I needed it most, who put up with all my crap, and who helped me get this thesis completed. To Shabby, who also wasted many late nights talking to me when I needed to take time off from this endeavor. And to Tal, who sat with me and helped me sort out ideas in my head, and always had encouraging words.

Last, but not least, I'd like to thank MIT, for making this possible, providing a wonderful educational experience, and being the hellhole it is (we all need some encouragement to finish our thesis).

Contents

1 Introduction **9**

2 What is Incremental Cryptography? **13**

2.1 Symbols and Terminology 13

2.2 Classical Cryptography 13

2.3 Incremental Cryptography 15

2.3.1 Efficiency 16

2.3.2 Data Representation 17

2.3.3 Privacy and History 18

2.3.4 Adversaries 20

2.4 Key Management 21

3 The Editor **23**

3.1 Cryptographic Transformations in Emacs 23

3.1.1 The Language 24

3.1.2 Usage of Modes 25

3.2 Picking the Right Scheme and Optimizing 26

4 Incremental Message Authentication Codes **30**

4.1 MACs 30

4.2 Incremental MACs 31

4.2.1 Symbols 31

4.2.2 Designing an Incremental MAC 32

4.3 Description of the Scheme 34
4.3.1 Initial Computation 34
4.3.2 Incremental Computations 35
4.3.3 Security Analysis 36
4.4 MAC-mode for Emacs 37
4.4.1 Internal Representation of Data 37
4.4.2 Operation of the Editor 39
4.4.3 Analysis of Performance 40
4.4.4 Other Ideas That Emerged 42

5 Public-Key Cryptography 44
5.1 Public-Key Cryptography 44
5.2 A Public-key Encryption Scheme 47
5.2.1 The underlying algorithms 48
5.2.2 The appropriately incremental algorithm 51
5.3 Analysis of Scheme 52
5.3.1 Efficiency 52
5.3.2 Security 53
5.3.3 Privacy 55
5.4 Implementation 55
5.4.1 The modules 56
5.4.2 API for Inter-Procces Communication 56
5.4.3 Analysis of Implementation 57

6 Secret Key Cryptography 60
6.1 Symmetric Encryption Schemes 60
6.2 An Incremental Block Cipher Symmetric Scheme 62
6.2.1 Description of Scheme 63
6.2.2 Analysis 67

7 Directions for Incremental Cryptography **72**
7.1 Other Related Ideas . . . 72
7.1.1 Incremental Hashing . . . 72
7.1.2 Incremental Digital Signatures . . . 73
7.1.3 Incremental Group Protocols . . . 74
7.1.4 Incremental Keys . . . 74
7.2 Possible Uses . . . 74
7.2.1 Filesystems . . . 75
7.2.2 Blinding . . . 75
7.2.3 Web Pages . . . 75
7.2.4 Electronic Cash . . . 76

8 Conclusion **77**

A Source Code **79**
A.1 MAC-mode . . . 79
A.1.1 MAC.el . . . 79
A.1.2 MAC-f1 . . . 104
A.1.3 MAC-f2 . . . 107
A.2 usage.el . . . 110
A.3 Incremental Public Key Encryption Code . . . 117
A.3.1 PKE.el . . . 117
A.3.2 PKE.c . . . 123
A.3.3 PKE-keygen.c . . . 137
A.3.4 PKE-decrypt.c . . . 143

List of Figures

3-1 Output of use-mode in emacs 28

4-1 MAC-list datastructure 38

5-1 Incremental Public Key Encryption Description 49

5-2 Appropriate Incremental PKE Description 51

6-1 Incremental OFB-ECB mode for a block cipher 64

List of Tables

2.1 Symbols used in document 14

3.1 Report on usage patterns for various buffers 27

3.2 Positions of inserts 28

3.3 Positions of deletes 29

4.1 Editor performance on append operations 41

4.2 Editor performance on random access edits (deletes, inserts, overwrites, etc.) 42

5.1 PKE-API specifications 59

Chapter 1

Introduction

Cryptographers often wonder: *"Why aren't more people using cryptography?"*. It certainly offers solutions to the problems of authenticity, security, freshness, and other issues dealing with digital data. However, while cryptography is slowly becoming more widely used and accepted, it still is not used nearly as much as cryptographers believe it should and could be.

An analysis of this question reveals several potential factors for this situation. Those include:

Political restrictions on the import/export of applications with useful cryptographic code. As it stands right now, companies in the U.S. are restricted from being able to sell software to clients outside the U.S. if there is support for 'strong encryption' within the program. Many other countries also have restrictions, varying from a complete ban on strong cryptography, to various types of import/export restrictions, to mandatory key escrow.

Difficulty of implementation and integration also worry many project managers. While the basic concept behind ciphers such as RSA and IDEA are simple, and the algorithms straightforward to implement, in practice, integrating those primitives into the software and still ensuring security is not simple. Problems that were not expected can creep in, and since the algorithm is only a small part of the protocol, the entire protocol still needs to be analyzed and verified

before it can be called "secure". This requires qualified people which are not easily available or affordable, and also time, which is an important factor in the design and implementation stages of software manufacturing.

Speed of the algorithm/protocol can make integration undesirable. Current encryption techniques are based on either mathematical principles, or else on the hope that the source plaintext is jumbled well enough to make unjumbling difficult without prior knowledge of a secret key. Message authentication codes (MACs) are usually based on hashes (which are fast to compute) and then encrypted with a scheme that is still somewhat slow. Until the schemes used become fast enough to not create delays for the user of the product, software manufacturers are going to worry about the speed issues of these schemes.

Ease of use issues for the user have also been problematic for cryptography. The problem facing the designers of cryptographic systems is a classical systems engineering problem: Users tend not to use things they feel uncomfortable using. In the case of encrypted e-mail, the delay while waiting for the encryption, and the need to actively select to encrypt and under what key and to which recipients, makes it an act most computer users avoid. While credit-card transactions on the web are probably as safe as (or safer than) purchasing on the street, people still appear to be worried about transmitting credit-card information on the World Wide Web. Secure HTTP pages are only made possible because it involves almost no work on the part of the user. *PGP*[18] has been available for years, yet few people use it since it is too complicated to figure out for many people.

Patents can also be a block to the use of specific algorithms. Inventors of the schemes want to be compensated for the research and idea, but not all companies believe the benefits of these schemes is worth the cost.

All of the above problems are related to why people or companies tend to not use cryptography in their software. In some cases, these explain why companies choose

to use weaker cryptography than they would otherwise employ. In other cases, these problems explain why companies actively chose to not use cryptography at all.

On the other hand, many of these same people recognize the many potential uses of cryptography, and want to have these benefits. These benefits can be grouped into three broad categories (extracted from [11]):

Privacy guarantees that only authorized persons (usually by giving them special keys) can use the data meaningfully. It can be guaranteed by encrypting the data in ways that prevent unauthorized users from reading it. Encryption, especially public-key encryption, is unfortunately normally *extremely* slow, and is one of the things that must be improved upon before it will be used in everyday practice.

Authentication proves to participants the identity of other participants. Authentication can be provided by concepts such as zero knowledge proofs and S-key passwords. This can be used for ensuring valid logins, for key management, and as a first step in authorization systems. Authentication schemes have also been used for making sure documents have not been altered.

Authorization is a scheme by which permissions can be granted to perform certain activities. It can be implemented using securely generated tokens or tickets, and unlike authentication based security, allows for partial anonymity.

The above categories use cryptographic primitives, including hashing, encryption (public and secret key), signing, and MACs, to achieve the desired properties. Some of the listed primitives, and how they are used, will all be discussed in the following chapters.

When choosing to provide the benefits, the previously mentioned costs are incurred. Many programmers and managers choose therefore to forego both the benefits and costs of cryptography. If the costs can be reduced, then it is likely that cryptography will be integrated into more applications and systems.

Incremental cryptography provides an approach for reducing the costs associated with ease-of-use, speed, and ease of integration of cryptographic methods into software products. The way incremental cryptography solves these issues is by embedding the actual primitives into the system, be it a file system, text editor, web browser, or one of many other systems in common use today. Furthermore, it can compute in the background while the object is being modified, instead of after (as with traditional cryptography), and takes only an amount of time proportional to the length of edits, which can help speed up the computation considerably. In order to have incremental cryptography succeed, we need schemes which allow for dynamically re-evaluating computations based on changes. With these tools, we can solve some of the problems associated with traditional cryptography.

Incremental cryptography does not tackle other issues, such as political issues and patents. One can hope that the political problems will be resolved favorably in time. One can also hope that patent issues can be resolved by the programmers and patent holders. However, in the worst case, the patents will expire after 20 years. So while this is a major delay (one that in the technological world feels like an eternity), it is nevertheless one that will be resolved eventually.

Chapter 2

What is Incremental Cryptography?

Incremental cryptography is an attempt to change the way most cryptographic transformations are computed. In a standard cryptographic transformation, the computation is performed on the completed document. Incremental schemes allow for the document to change slightly, and then they recompute the cryptographic transformation based on those changes in an efficient manner.

2.1 Symbols and Terminology

Throughout this paper, the symbols in table 2.1 will be used. Where appropriate, new symbols will be introduced in chapters, and explained within the chapter. If possible, similar concepts will attempt to use symbols similar to those in the table if possible (for example $E_K(D)$ represents an encryption of a document using key K).

2.2 Classical Cryptography

Under currently used cryptography, there are many basic primitives which achieve differing purposes. They can all be defined in terms of the following generic definition

Symbol	Meaning
$A \Rightarrow B$	a description of changes associated with modifying document A to become document B
$T_K(D)$	a transformation T (usually an encryption) using key K on document D
$C_K(D)$	a conjugate to transformation T (usually an decryption) using key K on document encrypted document D
P_j	The jth block of plaintext (usually the jth letter).
C_j	the jth block of ciphertext.
PPT $A()$	A probabilistic polynomial time algorithm $A()$

Table 2.1: Symbols used in document

of a cryptographic scheme[1]:

Definition 2.1 *A cryptographic scheme can be specified by the triple* $S = (Gen, T, C)$ *of probabilistic, polynomial time algorithms.*

- *Algorithm Gen is called the key generator. It takes as input* 1^k *(where k is the security parameter). It outputs a pair* (K', K'') *of keys to be referred to as the transformation key and conjugate key respectively.*
- *Both the transformation T and its conjugate C act on* Σ^**, using the corresponding key as additional input. We write* $T_{K'}(D)$ *to indicate the output of algorithm T on input D and key* K'*. We call* $T_{K'}(D)$ *a cryptographic form of D. For every* $D \in \Sigma^*$ *and every pair of keys* (K', K'') *possibly produced by* $Gen(1^k)$*, it is the case that*
  - $C_{K''}(T_{K'}(D)) = D$ *for all encryption and one-to-one systems.*
  - $C_{K''}(T_{K'}(D)) =$ **accept** *for other schemes (such as MAC's and hashes).*

This definition says nothing of security which is dependent on the type of operation. We note that a symmetric cryptosystem has $K' = K''$, while a public-key system has them set to be usually different.

[1]definition taken from [2]

Also, the definition does not take into account what the purpose of the scheme is. For each particular instantiation of a cryptographic primitive, a more specific definition exists and will be provided if necessary.

2.3 Incremental Cryptography

There are multiple possible types of changes to a document. They can all be grouped into three categories, however. Those are:

insertions lengthen the document by adding data somewhere. If the addition is at the end, we commonly refer to it as an **append** operation.

deletions shorten the document by removing data from within it. If the deletion occurs at the end, we commonly refer to this as a **truncate** operation.

modifications keep the length of the document the same, but change a block of data to a new value. Some systems implement a modify operation is a delete and insert operation.

Incremental cryptography takes a cryptographic scheme, and introduces an optimization to it, which lets us compute faster given a previous transformation. . We create an *incremental update* function which takes as input the kind of change made, and the old computation, and outputs a new computation based on those values.

Definition 2.2 *An incremental cryptographic scheme can be specified by the quad* $S = (Gen, T, C, Inc)$ *of probabilistic, polynomial time algorithms.*

- *Algorithm Gen is called the* key generator. *It takes as input* 1^k *(where k is the security parameter). It outputs a pair* (K', K'') *of keys to be referred to as the transformation key and conjugate key respectively.*

- *Both the transformation T and its conjugate C act on* Σ^**, using the corresponding key as additional input. We write* $T_{K'}(D)$ *to indicate the output of algorithm T on input D and key* K'*. We call* $T_{K'}(D)$ *a cryptographic form of D. For every*

$D \in \Sigma^*$ and every pair of keys (K', K'') possibly produced by $Gen(1^k)$, it is the case that

- $C_{K''}(T_{K'}(D)) = D$ *for the case of encryption schemes.*
- $C_{K''}(T_{K'}(D)) =$ **accept** *for the case of MAC or hash schemes.*

- *Algorithm Inc is referred to as the* incremental update algorithm, *or the* incrementor. *It takes as input the transformation key K', D (nominally a previous result of $T(old)$ where $old \in \Sigma^*$), and a list of changes from the old document: $[old \Rightarrow new]$. It outputs a value x such that:*
  - $C_{K''}(x) = [new]$ *for the case of encryption schemes*
  - $C_{K''}(x) =$ **accept** *for the case of MAC or hash schemes.*

The output of *Inc* is often fed back into *Inc* repeatedly to dynamically update the transformation of a document as it is edited.

2.3.1 Efficiency

Unlike with classical cryptography, incremental cryptography attempts to deal with the issue of efficiency from the perspective of the user as well as that of the algorithm.

There are several measures of efficiency for an incremental cryptographic computation:

- The first is the classical measure of order of growth. In the ideal case, we want the amount of time being spent computing to be proportional to some slowly growing function of the size of changes. The length of the document should ideally not affect the amount of time spent computing.
- The second measure is much harder to compute: We want the *perceived* delay to be smaller. For example, if this scheme is being used to dynamically create a signature for a document, we want the user to feel as if he is being slowed down less than if he were to simply compute the signature at the end. Given

the number of changes to the document may be very large, the total time spent dynamically creating the signature is probably much larger. However, since most of that is done using idle CPU time, the user never feels he is being delayed. On the other hand, computing entirely after finishing the document is completely user-visible.

- While most incremental schemes target the transformation algorithm, in practice, the conjugate is usually used without incrementing. What this means is that schemes should also keep in mind the efficiency of the conjugate algorithm relative to the speed of non-incremental schemes. This, as usual, depends on how often the conjugate is used relative to the transformation too.

- Finally, there are space considerations to keep in mind. Some schemes that will later be proposed take 60 times the storage space of non-incremental equivalents. While storage space is getting larger and cheaper, and network bandwidth is also increasing, it is preferable to have a scheme which requires approximately as much storage as its non-incremental counterpart.

The first, third, and fourth measures are easy to analyze and compute. The second needs to take into account how fast the machine is, how fast the user types, and what kind of changes are being made. To help analyze these things, a special mode was written for emacs which reports useful statistics, and is documented in chapter 3.

2.3.2 Data Representation

While the incremental editor is the main focus of discussion in this thesis, it is important to note that incremental cryptography can be used in many scenarios. As such, it does not always seem easy to decide how changes should be encoded. For example: say several people are working together on a software project. A common technique in this case is to use revision control such as CVS, which deals with integrating all the changes separately. Now say that these coders also want to submit message authentication codes for their source to be kept with each file. Using the concept of

'insert(pos, letter)' would be very dangerous in the MAC scheme, since this could easily cause problems when multiple people submit changes but have started with different versions of the file. A much better idea instead is to base the encoding of changes on a **diff**-like environment, which both encodes all the changes, plus gives enough context to figure out what to change where. Incidentally, this works well since CVS is also uses diff, and so those two concepts can be integrated well.

2.3.3 Privacy and History

There are many ways to achieve incrementality. For example, an incremental encryption scheme can be made by first encrypting a document, and then for every change made to the document, merely append to the encryption an encrypted description of how to change the document. While this works, it has several disadvantages. The first is speed related. While it is likely to be faster to compute the encryption, decryption (computing the conjugate function), especially if the list of changes is long, takes a considerable amount of time (proportional to the amount of change). Another is space related: since the document is appended with lists of changes, the size of the stored document grows disproportionately to the size of the actual resulting document. Both of those problems can be solved by regularly recomputing the function on the document from scratch, however.

The other real problem with the above scheme is a privacy issue. The history of a document is encoded into the transformation. Sometimes this is not a problem, and the above scheme can be used. Often, however, it is not desirable to send a signed e-mail to somebody which includes the fact that you deleted a line insulting him egregiously. A new worry of privacy caused by encoded history needs to be addressed, and for this reason, schemes should be designed to not have a history where possible.

Finally, another related problem to history encoding is the ability to tell whether the document was just created or whether it was incremented from another. One way to deal with the problem is to use datastructures to encode the document which do not reveal if an element was just added/deleted/modified, or if the datastructure was

created from scratch. Then, build the scheme using this datastructure to encode the document. Such a data structure is said to be oblivious[2]:

Definition 2.3 *Let $\mathcal{A} = (A, \Sigma_A)$ be a probabilistic data structure, and let $\prec$ be a congruence relationship over $\mathcal{A}$.*

We say that $\mathcal{A}$ is oblivious with respect to $\prec$ if for any two terms t^1 and t^2, if $t^1_A \prec t^2_A$ then t^1_A and t^2_A define polynomial-time indistinguishable probability distributions.

This suggests that the data structure representing the actual document through $T()$ is hard to tell from the data structure formed by *Updt*.

We want the data structures to be oblivious so that the adversary will be unable to learn how the structure got formed. Since any state corresponds to all possible ways of achieving that representation, all paths to this data and its corresponding representation are as likely. Therefore, no history can be extracted from the computation.

Another side-effect of incrementing is the leaking of external information. While the transformation can be designed to not encode a history of changes, it may still leak some information (for example, whether this is a document computed from scratch or whether it is a document which has been updated at least once). Ideally, therefore, we can demand *perfect privacy*:

Definition 2.4 *Informally,* perfect privacy *is achieved when the transformation of a document is indistinguishable from a documented created by using the* Increment *algorithm on a previous transformation:*

$\forall M$ polynomial time probabilistic Turing machines, $\forall (D, E) \in \Sigma^l$ possible documents (with l being the length of the document), given $Y = T_{K'}(D)$ and $Z = Updt(T_{K'}(E), (D \Rightarrow E))$, $\forall Q()$ polynomials, we say that a cryptographic scheme achieves perfect privacy if $\exists k_0$ s.t. $\forall k > k_0$:

$$Pr[M(Y) = accept \ and \ M(Z) = reject] < \frac{1}{Q(k)}$$

[2] This definition is extracted from [12], which also discusses many other aspects for oblivious data structures, and probabilistic data structures

where the probability is taken over the random bits in $T_K()$ and $Updt()$, and the choices of D and E.

An oblivious data structure (definition 2.3) is guaranteed to achieve this property, but there may be other techniques. This definition can instead suggest that Z and Y be identical, which is a stronger requirement on the scheme. Several of the schemes that will be proposed achieve this requirement anyway.

To make matters worse, even if there is no history, there is a worry that seeing two different transformations will yield some information about the difference between the two plaintexts:

Definition 2.5 *For all possible $(r_1, r_2, D, E) \in \Sigma^l$,* indistinguishable privacy *is achieved when, for all polynomial time probabilistic Turing machines M, given $Y = T_{K'}(D)$ and $Z = T_{K'}(E)$, $\forall Q()$ polynomials, $\exists k_0$ s.t. $\forall k > k_0$:*

$$Pr[M(Y, Z, (D \Rightarrow E)) = accept \text{ and } M(Y, Z, (r_1 \Rightarrow r_2)) = reject] < \frac{1}{Q(k)}$$

where the probability is take over the choice of r_1 and r_2, D and E, and the random bits of $T_K()$.

This final requirement ensures that even after seeing an original plaintext and its transformation, seeing the transformation of another plaintext will not reveal to us any information about what changed. This is a very strict requirement, and is not met by all schemes.

With this final requirement, we have all that we need to analyze our schemes in terms of privacy.

2.3.4 Adversaries

In the non-incremental document scheme. A chosen plaintext or ciphertext attack assumes the adversary can request that one of the above be used. A passive adversary does not have that advantage, and in many systems, only a passive adversary

can really exist. However, incremental cryptography introduces an adversary that is passive, and yet might receive a large choice of related messages merely because over time, the document is changed. An active adversary, on the other hand, can use weaker attacks and still learn more than he used to, because he no longer needs to submit complete documents for encryption/decryption or signing. Instead he can submit changes to the documents and original computations to observe what happens. This can be especially useful in a scheme that has history, where submitting the two documents independently will yield a different computation than submitting the first and instructions on how to reach the second.

All of the above techniques may be used, and should be watched out for, when designing incremental cryptography schemes. The adversary has at hand all the classical attacks, plus new ones dealing with the fact that the document is dynamic, and so is the transformation.

In many cases where analysis needs to be performed, it is very important to know how much one can expect the passive adversary to see. In some situations, where incrementality is just being used to make computations of signatures for a single document dynamically, the adversary is likely to only see the final computation. On the other hand, if the incrementality is being used in communications between two people, the passive adversary will see many pairs of related messages. Ideally, we want to be secure against the latter. Usually, however, we can't guarantee that, but can usually be more secure than the former.

2.4 Key Management

One more area which will not be tackled in this thesis but that must be understood well is that of key management. There are many schemes in use today to facilitate key management, from things such as kerberos [17], to key rings for PGP [16]. The basic problem is that we wish to make the scheme convenient for the users, and at the same time offer security which is based on their knowledge (i.e. keys).

In the ideal system envisioned by this thesis, the user will be asked for one pass-

word at login time for authentication purposes, and then have the system perform these computations automatically and silently for him. For example, at login, the user will be asked for a username and password, which will then be the key used to encrypt all his files on the filesystem silently (and using incremental techniques to prevent major slowdowns).

Chapter 3

The Editor

In order to see how effective these ideas are, and in order to see what else can be learned from using an actual system which embeds incremental cryptography, we implemented some of the ideas in an editor. This chapter will discuss specific details, and then later chapters will discuss the particulars of the implementation with respect to the scheme used.

3.1 Cryptographic Transformations in Emacs

The editor we chose to put implement these transformations within was Emacs. The choice was relatively easy, as it was one of the few editors that runs on most operating systems, has an internal control language which lets one access almost everything the editor does, and is freely available. A feature which came with this was the fact that the code (even byte-compiled) would run on any platform. We used Emacs version 19.34, which had several features not available in previous versions, and so the code will not run on older versions of emacs.

Emacs views the files it is editing as 'buffers', and can have multiple buffers being edited at a time. Also, buffers do not have to correspond to any file, but can instead be generated by programs.

3.1.1 The Language

Emacs is programmed in Elisp, which is a Common Lisp derived language. It has many convenience functions for handling concepts such as buffers, files, and other objects related to the editor. This language provides access to the buffer, and lets us track changes made to it. We use this feature to provide two minor modes for it.

Modes

A minor mode in emacs is a function that is enabled or disabled on a per-buffer basis. Multiple minor modes can be activated in any buffer, each doing some special task. Common tasks include automatic indentation of text, highlighting of areas, and automatically wrapping lines when they get too long.

Minor modes operate in conjunction with Major modes. The difference is that only one major mode can be active in a buffer at any one time. Common major modes include News Mode (Gnus), Rmail Mode (email), and Outline Mode. By choosing to implement our feature as a minor mode, we have effectively allowed not only for the creation of an encryption (for example) in a buffer, but also allowed us to automatically encrypt our mail or news transactions. Hooks need to be added to allow these computations to be transmitted along with or instead of the document, but these are usually fairly easy to do.

Speed of Execution

One unfortunate problem with elisp (and with most lisps) is their inherent slowness. Elisp offers features such as garbage collection, but at a cost. The original implementation of the MAC mode had the DES coded written in Elisp. It was incredibly slow and inefficient.

The solution is to use a feature Emacs provides which allows for asynchronous communication with subprocesses. Most of the time-critical code could then be coded in C or assembly, while the main buffer-related functions were left in elisp. While this did mean that the editors would no longer automatically run on any platform

(they need recompilation for new platforms), the modes ran much faster (by at least a factor of twenty).

Then, a protocol was designed for talking with the subprocess, and communications worked back-and–forth until a final computation was achieved. Interesting to note, the approach used for the MAC mode (see chapter 4) did most of the work in elisp, collecting data structures internally, and using subprocesses only to do the DES computations. On the other hand, for the PKE mode (chapter 5), the work was mostly done in the subprocess, with the elisp just responsible for informing the subprocess what is going on. These two approaches had different benefits:

- Putting most of the work in emacs let us optimize on using idle-time. Emacs can tell when the user is not doing anything at the moment, and use that to dedicate as much of the CPU towards incremental computations. Then, when the user returns to working, it slows down on the computations, and lets the user get his edits done (leaving the complex computations until later).

- Putting the work in subprocesses lets us be much more modular, allowing different subprocesses to do varying things for the same mode (for example, the Public Key Encryption mode could be changed to do symmetric encryption without any major changes to the elisp code). Also, the same subprocess can be used by other programs (perhaps another editor or maybe a web browser) to generate incremental encryptions for them.

As can be seen, both techniques have their advantages and disadvantages. However, from a user's perspective, after trying out both, it's pretty clear that the former feels a lot more efficient.

3.1.2 Usage of Modes

Although ideally, I would like all of this to be automatic, the current implementation requires some intervention from the user and the administrator.

First of all, it needs to be installed, both the binaries and the elisp code. Then, emacs has to be configured to know where the code is and what to do with it. Finally,

while the mode is automatically activated for any files which have had transformations computed on the previously, the default (which can be changed), is not to activate it for all buffers. Finally, the user needs to also intervene: since key management isn't part of the system, the user may be prompted for a username or the value of a key.

The above only need to happen once, at the beginning of a session, but are still user-visible. Ultimately, with proper key-management techniques, it may become much simpler to use and handle, and there will be no user-intervention required.

3.2 Picking the Right Scheme and Optimizing

There are many incremental schemes that will be proposed in this thesis, each targeting a specific aspect of cryptography. Which one to use is usually a matter of requirements, but even after selecting the scheme, there is a lot of fine-tuning that can be done to increase performance. Things such as block sizes, or ordering, or amount of time to wait in idle mode before actually beginning to process are all variables that need to be set as appropriate.

In order to analyze this, we need to have some way of judging what kind of editing is being done. Therefore, a small analysis mode for emacs was created which observes and collects relevant statistics on the kinds of changes made to a buffer. While this program is in now way a complete measure of usage, it can be very helpful in determining what to set specific variables to within schemes, and also sometimes helps determine which variant of a scheme to use.

The source code is in A.2, but the basic idea is:

total changes are collected. This is the sum of all the types of changes that were performed on the buffer.

inserts are counted, and then a percentage is computed based on the total number of changes: $(inserts * 100)/changes$.

deletes are counted, and like for inserts, a percentage is computed.

Table 3.1: Report on usage patterns for various buffers

type of edits	percent of changes:			percent of inserts that are appends	percent of deletes that are truncates
	inserts	deletes	modify		
composing a chapter	86 %	14 %	0 %	95 %	85 %
proofing a text file	61 %	38 %	0 %	0 %	1 %
writing code	73 %	27 %	0 %	26 %	12 %

modifications are classified as any buffer change which causes no length changes. A percentage of total changes is again computed.

appends are counted as inserts made at the end of the buffer. Their percentage is computed thus: $(appends * 100)/inserts$.

truncates are counted as deletes made at the end of the buffer. Their percentage is computed in a similar manner to appends.

subdivisions of insertions are then computed. Each insert is analyzed for its position (which quarter of the buffer it occurred in). These values are then converted to percentages of total inserts.

subdivision of deletes are then also computed in a similar manner to those for inserts.

Figure 3-1 is the output of calling `use-report` on the buffer being studied. These were current numbers while I was making a few modifications to this chapter.

The statistics in tables 3.1, 3.2, and table 3.3 have been collected for different types of editing of a buffer, for reference and example of what kinds of changes might be used. The scheme that should be used for the above should be optimized for appends with many short delays in between. Furthermore, it should use relatively large blocksizes, since most edits seem to occur in the end.

Given the above statistics, it can readily be determined that for a text composition, most of the changes occur at the end, which suggests schemes that are optimized for changes in the latter parts of the document should be chosen. Also, in all these cases,

```
-------------------------------------------------
   Use-mode report for buffer chap3.tex
-------------------------------------------------
Total number of changes made :    24
                     inserts :    50 %
                     deletes :    50 %
               modifications :    0 %
                     appends :    0 %
                   truncates :    83 %
 the insertions occurred in:
                 1st quarter :    0 %
                 2nd quarter :    16 %
                 3rd quarter :    0 %
                 4th quarter :    83 %
 the deletions occurred in:
                 1st quarter :    0 %
                 2nd quarter :    16 %
                 3rd quarter :    0 %
                 4th quarter :    83 %

 The editing of the buffer took 3802 seconds.
 During the editing of the buffer, it was idle for
     between 1/2 sec to a sec :   44 times
      between 1 sec to 2 secs :   24 times
  between 2 secs to a 10 secs :   32 times
                 over 10 secs :   4 times
------------------------------------------------
```

Figure 3-1: Output of use-mode in emacs

Table 3.2: Positions of inserts

type of edits	$1^{st}quarter$	$2^{nd}quarter$	$3^{rd}quarter$	$4^{th}quarter$
composing a chapter	0%	0%	1%	99%
proofing a text file	12%	44%	31%	13%
writing code	3%	12%	8%	77%

Table 3.3: Positions of deletes

type of edits	$1^{st}quarter$	$2^{nd}quarter$	$3^{rd}quarter$	$4^{th}quarter$
composing a chapter	0%	0%	0%	100%
proofing a text file	24%	35%	28%	13%
writing code	15%	3%	5%	77%

modify operations are nonexistent. This is probably mostly due to the way in which editing is done within emacs. However, there are situations where modify operations might be the most common type of operation (for example, in a database where keys are constant, but data values keep changing).

There is also another aspect to be examined: the amount of delays between changes. In the case of a text buffer, it is basically a function of how fast the user types, and what kind of breaks he takes to think. Breaks of longer than ten seconds are grouped into one category, since it is assumed that in more than ten seconds, it is probably possible to compute whatever computation is needed from scratch, and so the buffer will be assumed to be computed.

With the ability to analyze these statistics, the scheme chosen can be optimized on a per-user or per-application basis.

Chapter 4

Incremental Message Authentication Codes

Message authentication codes are used to demonstrate the authenticity of a message. They usually combine a tagging function which generates the MAC, and a verification function that ensures the MAC is associated with the document.

4.1 MACs

A MAC is defined thus[1]:

Definition 4.1 *A message authentication scheme is a triplet $(KGen, Tag, Vf)$ of probabilistic, polynomial time algorithms. The first algorithm is the* key generator, *and it takes as input a security parameter 1^k and yields a key $K = KGen(1^k)$ which is used in the other algorithms. The second algorithm is the* tagging *algorithm or the* MACing *algorithm and it takes as input a message M, yielding a MAC $\sigma = Tag(K, M) \equiv Tag_K(M)$. The last is called the verification algorithm, and it takes as input the key and the MAC. We require that if $\sigma = Tag_K(M)$, then $Vf_K(M, \sigma) =$* **accept**, *and that:*

[1]Definition provided by [4]

$\forall PPTA()$ $\forall M \in \Sigma^*$ possible messages, and $\sigma = Tag_K(M)$ and $\forall P$ PPT algorithms, $\exists k_0$ s.t. $\forall k > k_0$:

$$PR\Big[A(M,\sigma) = M' \quad \textit{such that} \quad Vf_K(M',\sigma) = \textbf{accept}\Big] < \frac{1}{P(k)}$$

We instantiate this scheme for MACs in the case of deterministic algorithms.

Definition 4.2 *A message authentication code is a deterministic, polynomial-time computable function MAC such that (MAC, Vf) is a message authentication scheme, where $Vf_K(M,\sigma)$ is defined to compute $MAC_K(M)$ and* **accept** *if and only if this value is equal to σ.*

This is how the classical definition of a MAC scheme works, but we need to allow for incrementality.

4.2 Incremental MACs

Now we explain a scheme[2] which creates a MAC for a document using any block cipher.

4.2.1 Symbols

In the following definitions, the following notations is used:

1. A document D is viewed as a sequence of n blocks, each b bits long, let $B_b = \{0,1\}^b$ denote the domain for each block, and B_b^n be the domain of a document.
2. $D[i]$ is the i-th block of $D \in B_b^n$.
3. IncM is the incremental scheme being used to update the MAC of a document. It takes as input $(K, MAC_{old}, (old \Rightarrow new))$. It corresponds to *Inc* in the generic incremental cryptographic scheme definition.

[2]due to Bellare, Goldreich, and Goldwasser and detailed in [3]

4. A change in the document, and the corresponding change in the MAC, is denoted by the following requests:

   - A replacement is denoted by $\text{IncM}(D, mac, \text{repl}, (j, m))$. In this request, $D[j]$ will now contain m and the rest of D will remain unchanged. We let $D\langle j, m\rangle$ be shorthand for the above, where m is the new block.
   - A deletion request is denoted by $\text{IncM}(D, mac, \text{del}, j)$. In this request, $D[j]$ is removed from the sequence, so $D[j-1]$ is followed by $D[j+1]$ in the document. Stated otherwise, the new document now looks like:

$$D = \begin{cases} (D[1]..D[n]) & \text{if } j = 0 \\ (D[0]..D[j-1] \cdot D[j+1]..D[n]) & \text{if } 0 < j < n \\ (D[0]..D[n-1]) & \text{if } j = n \end{cases}$$

$$n = n - 1$$

   shorthand for the above is $D\langle -j\rangle$.

   - An insertion request is denoted by $\text{IncM}(D, mac, \text{ins}, (j, m))$. In this request, the j-th block is now followed by a new block containing m. Otherwise stated:

$$D = \begin{cases} (m \cdot D[0]..D[n]) & \text{if } j = -1. \\ (D[0]..D[j] \cdot m \cdot D[j+1]..D[n]) & \text{if } -1 < j < n. \\ (D[0]..D[n] \cdot m) & \text{if } j = n. \end{cases}$$

$$n = n + 1$$

   shorthand for the above is $D\langle +j, m\rangle$.

4.2.2 Designing an Incremental MAC

We extend the definition of a message authentication code to allow for incrementality. We introduce independence (as suggested in [2]) of the security parameter k, the number of blocks in the message b, and the size of each block n.

Definition 4.3 *a family of message authentication codes computing functions is defined by the triple $\mathcal{M}$ = (Mgen, Meval, Vf) of algorithms.*

- *The PPT generator Mgen takes as input 1^k, 1^b, 1^n, and returns a string K (which is used as a key). —K— is related to a polynomial in k.*
- *The PPT evaluator Meval takes K and a document $D \in B_b^n$, and outputs a k bit string that is the message authentication code for the appropriate document.*
- *the PPT verifier Vf takes a k-bit string σ, the key M, and a document D, and outputs* **accept** *if and only if σ was generated by Meval working with M on D.*

We now need to create an update function that will allow us to incrementally change the MAC without recomputing from scratch. This is achieved via IncM which turns the MAC of D into the MAC of $D\langle j, m\rangle$, $D\langle -j\rangle$, or $D\langle +j, m\rangle$ depending on the change desired. We use ideas presented in previous papers to extend MAC computing functions.

Definition 4.4 *Let $\mathcal{M}$ = (Mgen, Meval, Vf, IncM) specify a family of MAC computing functions. We say that IncM is an update algorithm for $\mathcal{M}$ with running time $T(\cdot,\cdot,\cdot)$ if*

$$\forall k, b, n, \forall M \in [Mgen(1^k, 1^b, 1^n)]\ \forall j \in \{1, ..., n\},\ \forall m \in B_b,$$

if mac = MEval(M, D) then it is the case that:

- *IncM$(M, D, mac, repl, (j, m))$ halts in $T(k, b, n)$ steps with an output that is polynomial-time indistinguishable from the output of MEval$(M, D\langle j, m\rangle)$.*
- *IncM(M, D, mac, del, j) halts in $T(k, b, n-1)$ steps with an output polynomial-time indistinguishable from MEval$(M, D\langle -j\rangle)$.*
- *IncM$(M, D, mac, ins, (j, m))$ halts in $T(k, b, n+1)$ steps with an output polynomial-time indistinguishable from MEval$(M, D\langle +j, m\rangle)$.*

We call the IncM-*augmentation* of $\mathcal{M}$ =(Mgen, Meval, Vf) the quad $\mathcal{M}^+$ =(Mgen, Meval, Vf, IncM).

Note that in this scheme, the verification algorithm is not incremental.

4.3 Description of the Scheme

With the above requirements and definitions, we now describe the actual scheme that was implemented:

4.3.1 Initial Computation

The scheme proposed in [3] works in the following manner:

There is a key $M = (k_1, k_2)$ which is held by both the MAC generator and MAC verifier in secret. Furthermore, there is a pad-generating function `rand` which adds a randomizer to every block in the message (i.e. given a string σ, it returns $\sigma \cdot r$ where r is a random value). There are two pseudo-random permutation functions, f_1 and f_2, which take as indexes (keys) k_1, k_2 respectively and are used to compute the MAC:

To compute the MAC for message $D = (D[1]..D[n])$ we prefix it with a special start block $D[0]$ and postfix with an end block $D[n+1]$, yielding $D = (D[0]..D[n+1])$. Then, for each block, we add a randomizing pad by calling `rand()` with the value in each respective block. `rand()` returns a random number of bits that are necessary to fill in the pad (this is a variable). This procedure yields a series of $n + 2$ blocks containing the data and random pad for each block: $R = (R[0]..R[n + 1])$.

Now, we use an idea proposed in [3] for generating MACs, which is to block-cipher-chain the respective R's in the following manner:

$$mac = f_2(\bigoplus_{i=0}^{n} f_1(R[i], R[i + 1]))$$

This is the initial MAC, and from this point on, all changes to it are computed incrementally depending on the type of change. Note that an incremental computation in this case yields a MAC that is exactly the same as computing it again from scratch

(i.e. there is no history of changes encoded in the MAC), this therefore achieves the requirements of *perfect privacy* (definition 2.4).

4.3.2 Incremental Computations

There are two approaches to computing the new MAC. The first method assumes that space is not a concern and saves all subcomputations for the document. The second method trades a small amount of computation time for space-efficiency, and only stores the random pads and the final MAC. The second, however, also requires that f_2 be invertible if the secret key is known. This may or may not be problematic (in our implementation, it was not, since f_2 was invertible). Here are the ways to deal with changes to the document under both models:

Modifying a block

A change in the data of only one block is easy to deal with. $D\langle j,m\rangle$'s MAC is computed by the following technique:

1. Under this method, all the block pairs already have their f_1's computed, so all that is needed is to find the two blocks whose f_1's are affected (block pairs $(D[j-1], D[j])$ and $(D[j], D[j+1])$) , and update them. Then, recompute the final sum.

2. If we do not have the local computations stored somewhere, then we need f_2 to be reversible, and we do the following:

$$mac = f_2\Big(f_2^{-1}(mac) \oplus f_1\big(D[j-1], D[j]\big) \oplus f_1\big(D[j], D[j+1]\big) \\ \oplus f_1\big(D[j-1], \mathbf{rand}(m)\big) \oplus f_1\big(\mathbf{rand}(m), D[j+1]\big)\Big)$$

Adding a block

$D\langle +j,n\rangle$'s new MAC is computed thus:

1. Under this scheme we compute the f_1's associated with the pairs that touch the block being added (those being $(D[j], m)$ and $(m, D[j+1])$). Then, we XOR

these new values with the rest of the f_1's already computed to get a new hash which we call f_2 upon.

2. For this scheme, again we perform some more work, and the formula we use is:

$$mac = f_2\Big(f_2^{-1}(mac) \oplus f_1\big(D[j], D[j+1]\big) \oplus f_1\big(D[j], \mathtt{rand}(m)\big) \oplus f_1\big(\mathtt{rand}(m), D[j+1]\big)\Big)$$

Deleting a block

The final type of change that can be made is a deletion of a block. to compute the MAC for $D\langle -j\rangle$, the following is done:

1. for a scheme where everything is stored, we just need to recompute one pair, the $(D[j-1], D[j+1])$ pair, and throw away two old ones.

2. for the less memory-intensive scheme, we compute the following:

$$mac = f_2\Big(f_2^{-1}(mac) \oplus f_1(D[j-1], D[j]) \oplus f_1\big(D[j], D[j+1]\big) \oplus f_1\big(D[j-1], D[j+1]\big)\Big)$$

4.3.3 Security Analysis

A formal analysis of XOR-based MACs was performed in [5]. There they demonstrated that MACs generated by XOR-ing in the manner performed above can be considered secure[3]. Since all that was done here is turn that MAC into an incremental one, the security still holds, and this was analyzed in [3].

[3]There are assumptions made about the permutation, see the paper for a complete analysis

4.4 MAC-mode for Emacs

To see how well this concept works when applied to a commonly used system, a minor mode was written for emacs to do exactly that. The minor mode implements the scheme described above, using asynchronous communication with an external program to achieve maximal efficiency. The program has three parts:

MAC.el (A.1.1) is the actual elisp minor-mode definition code. It is responsible for deciding which buffers to compute MACs for, and is also responsible for automatically saving and loading the MAC data when a file is loaded.

MAC-f1.c (A.1.2) is an auxiliary program which received the secret f_1 key, and then receives the data on the two blocks, outputting the computation of $f_1(B_1, B_2)$, which is then used by **MAC.el** to do its work.

MAC-f2.c (A.1.3) is the second auxiliary program. Much less often called, it is responsible for performing the final encryption, and communicates with **MAC.el** in the same way as above.

4.4.1 Internal Representation of Data

When a document is first loaded, or `MAC-mode` is activated on a buffer, there is a check to see whether the data representing the buffer, including the computation history, is available in an auxiliary file (saved as {filename}.sum). If it is there, it is loaded in and used. Otherwise, the basic datastructure is created thus:

- First, it breaks the buffer (usually an empty one) into blocks of size `MAC-blocksize`. This value can be set anywhere from 1, which indicates 1 byte, to however large the largest string emacs will allow is (usually the size of a page in the operating system).

- Next, it generates a pad of size `MAC-security-padding` bytes for each block. The same restrictions apply as to the data in each block.

- It marks every f_1 value in the blocks as `'nil`, which is reserved to indicate that the checksum has not been computed yet.

- Finally, it adds an end-block to the linked list, which is used to indicate there are no more blocks in the list.

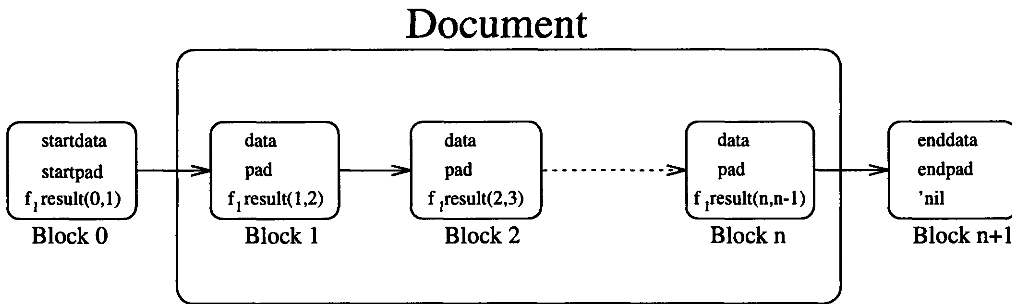

Figure 4-1: MAC-list datastructure

The above steps form the *MAC-list* datastructure (see Figure 4-1). Elisp linked-list (car/cdr pairs) with the properties that the `MAC-string` element for each item in the list holds a `MAC-blocksize`-length string which reflects the contents of the buffer. If all the `MAC-strings` were laid end-to-end, the exact contents in the buffer will result. The `MAC-pad` holds the random pad (that which is gotten from the `rand()` function. Finally, the `MAC-sum` holds the result of the f_1 function called upon the `MAC-string` and `MAC-pad` of the current block and of the next block. It may also hold the special value of `'nil` to indicate that the computation has not been performed yet.

The above implies that the end-block's sum is always 'nil, and that the start-block's sum is variable (so it cannot be made into a constant, and is a different datablock for each buffer). Also, there is one further problem, which lies in the fact that the last data block may not always contain exactly `MAC-blocksize` bytes of data. This is acceptable from a security point of view, but in the implementation, this is restricted to happening on the last block only. (See section 4.4.4 for a discussion of variable sized blocks, and other ways to improve this).

4.4.2 Operation of the Editor

Now, we have a datastructure to represent the buffer. From this point on, one of two things may happen: Either the machine will be idle, or the user will be typing something. If the machine is idle, the mode will attempt to find f_1 values that have not been computed. If the user is typing, then it will attempt to make the *MAC-list* datastructure reflect the status of the buffer.

If the machine is idle, then the minor mode will utilize this time to update the MAC dynamically. Since most users take breaks while typing, and since most users don't type that quickly anyway, most of the CPU time is spent in this mode. During this time, the list is traversed until a block is found whose `MAC-sum` is set to `'nil`. Once a block is found that has this property, its sum is computed based on the data stored within the current as well as the next block. If all blocks are computed correctly, then the buffer's MAC is computed and held until the time when the user requests it, or else the buffer changes again.

If instead of leaving the machine idle, the user types something which causes changes to the buffer, the internal *MAC-list* is changed to reflect those changes. There are many possible types of changes, and since the algorithm is designed to work in blocks, each of these changes can lead to several possible types of changes in the structure. Any change is reported as a $(startpos, endpos, newlen)$ triplet, which is all that is necessary to figure out what has changed:

- A modification usually occurs when in *overwrite* mode. In this mode, anything that is typed is typed over previous characters. It can, however, also occur in some specific modes where some text that is being typed is replaced with different text (for example, automatic capitalization). In this case, no length change occurs ($endpos - startpos = newlen$). This of course means that the *MAC-list* structure doesn't change, although the data within the respective blocks, as well as the respective f_1 values, do. This is easy to deal with.

- A deletion can occur due to a delete or backspace key. Or on a larger scale, due to a cut operation (among others). This can be more tricky, as the dele-

tion modifies the length of data in a block, and in some cases, can delete an entire block or more. The implementation can only use the algorithm's delete operation when an entire block is removed. In all other cases, the length of a block changes, and so in the worst case, characters from further blocks need to be shifted into the current block to fill it to the right length (and so on for the further ones down). Deletions early on in a document can lead to the entire document's MAC being recomputed from scratch.

- Insertions are very similar to deletions. They can be brought about by almost any editing command in emacs, as well as automatically due to things like C-mode. Like deletions, they cause block data lengths to change, and so require either pushing data forward through blocks, or in lucky cases, the new data fits completely into a new block between two other ones.

If any block changes, its data is set to reflect the new data, but the sum is left as a `'nil`. The cryptographic computations are not done until such time as the user isn't typing anything, or the user forces the computations (for example, by asking for the MAC, or saving out the file).

An attempt to save the file causes the editor to go through the entire structure to make sure it is correct (all data blocks contain the right number of bytes of data from the buffer, and all f_1 computations have been performed). Then it computes the xor of all the f_1 values (if that has not yet been done in idle time), and finally, computes the f_2 of the xor. The *MAC-list* datastructure, as well as the result of the f_2 (the MAC) are written out to the filename with a '.sum' postfix.

4.4.3 Analysis of Performance

The above scheme was implemented, and tested using a SparcStation 1 and on a pentium machine running NetBSD/i386. The emacs used was emacs.19.34.1. For measurement purposes, we set the f_1 and f_2 to be DES, although there are many other alternatives, such as IDEA and combinations of MD5 and DES which would have

worked as well. The typing rate of the person using the software was approximately 80 words per minute.

A basic comparison of apparent performance was done, comparing the integrated MAC evaluator with differing parameters. Since the timing here isn't usefully measured in CPU cycles or seconds, it is hard to compare it to a scheme which computes completely at the end. Instead of measuring using time, we measure what percentage of the document is precomputed by the time the user wishes to save the buffer. Figures 4.1 and 4.2 detail the performance on a Sparcstation 1. For the append, a buffer was simply typed into continuously at a specific rate. For the random edits, three different speed typing was used to edit a buffer, using cuts and pastes, overwrites, insertions, and other operations done to files such as code. It was started with a file that had the MAC fully computed for, and then checked to see how much of the buffer was uncomputed at the end. Although this measure is very dependent on the exact types of changes, the typer was not aware of where boundaries were, and inserted, deleted, and overwrote throughout the buffer.

parameters	performance
blocksize = 1 pad size = 1 80 WPM	Editor was straining, almost never managing to find enough time to compute f_1. Approximately 3% of the blocks got computed.
blocksize = 1 pad size = 1 30 wpm	could mostly keep up, with only about 10% of the blocks uncomputed
blocksize = 64 pad size = 16 80 wpm	Editor would have been happier with a few breaks in between, but managed to get 40% of the blocks computed by the time a save was requested.
blocksize = 64 pad size = 16 30 wpm	80% of the buffer was computed

Table 4.1: Editor performance on append operations

As can be seen in the above data, the kind of input, as well as the person who is typing, and the choice of variables, all control the efficiency of the scheme. Furthermore, since this scheme is implemented in elisp,it can be improved upon considerably

parameters	performance
blocksize = 64 pad size = 16 80 WPM	Only 5% was computed
blocksize = 64 pad size = 16 30 WPM	26% of the buffer was computed

Table 4.2: Editor performance on random access edits (deletes, inserts, overwrites, etc.)

by integrating it using C into a different word processor. While it may seem that only order of growth of the algorithm is important, it turns out that small optimizations can make major differences in how well the scheme operates with the user.

One final thing that deserves some analysis is the verification scheme. Theoretically, verification should be as fast as generation from scratch, and it is. Incrementality as of yet does not offer any speedups in the verification scheme (although should one choose to use diffs, they are smaller to verify). However, since elisp is rather slow, the scheme itself, when compared to native-c-code programs such as PGP, appears much slower. In theory, the scheme should not be any slower (as was demonstrated in previous papers), but due to the lack of good popular text editors with free source code in C, it was difficult to test.

4.4.4 Other Ideas That Emerged

After trying it out for a while, we had noticed several interesting problems, and had alternative approaches to solving them.

- The constant blocksize (initially envisioned to be 1 byte by the authors) can lead to several slowdowns. If it is set at just one character, then there are a lot of f_1's computed per document. If it is set at a large number, then the probability of causing incomplete blocks in the middle of a document are increased (leading to the need to adjust data further down, and invalidating all the f_1's further

down the list). It turns out that depending on the kind of editing that is done more often, a different model should be used. Larger blocksizes lent themselves very well to appending operations, and database-style operations that dealt with data in blocks. One byte blocksizes were better for random-access edits.

- Another idea noticed was that there really was no need for a constant blocksize. Allowing the size of the data in a block to vary stops all the problems caused by data being unaligned. Since nothing in the proof of security contains anything requiring the blocks to use constant lengths, we can allow for blocks to have varying sized data.

- These ideas can also be merged with predictive/adaptive algorithms to allow the computer to try and *guess* what the user will type, and precompute some blocks.

Chapter 5

Public-Key Cryptography

This chapter will deal with applying incremental cryptography to the commonly used cryptographics schemes known as public-key cryptography. It will offer a scheme and analyze it in terms of efficiency and security.

5.1 Public-Key Cryptography

Talked about initially in [8], private key encryption is often compared to private key encryption. It is defined in the following manner[1].

Definition 5.1 *A* public-key encryption scheme *is a triple $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ of probabilistic polynomial time Turing machines, together with an indication of key length k, satisfying the following conditions:*

key generation algorithm: *a probabilistic expected polynomial time algorithm $\mathcal{G}$, which on input 1^k produces a pair (e, d) where e is the public key and d is the private key: $(e, d) \in \mathcal{G}(1^k)$.*

encryption algorithm: *a probabilistic polynomial time algorithm $\mathcal{E}$ which takes as input a security parameter 1^k, the public key e from $\mathcal{G}(1^k)$ and a string $m \in \{0,1\}^k$, and outputs $c \in \{0,1\}^*$ which is the encryption of m using e as the key.*

[1] adapted from [4] pages 67–68

decryption algorithm: *a probabilistic polynomial time algorithm* $\mathcal{D}$ *that takes as input a security parameter* 1^k*, a private key* d *from* $\mathcal{G}(1^k)$*, and a ciphertext* c *from* $\mathcal{E}(1^k, e, m)$*, and produces* m.

We also need to explain the security requirements of this scheme, so, as for symmetric encryption, we will use the indistinguishability requirement[2], although it has been proven to be equivalent to the semantic definition sometimes used instead.

We define it such that it is hard to find any two messages whose encryptions are distinguishable:

Definition 5.2 *A public-key cryptosystem* $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ *is* polynomial-time indistinguishable *if for every message length* l*, and for every polynomial* Q*,* $\exists k_0$ *s.t.* $\forall k > k_0$*:*

$$Pr\Big[A(1^k, e, m_0, m_1, c) = m \quad | \quad (e,d) \xleftarrow{R} \mathcal{G}(1^k); \{m_0, m_1\} \xleftarrow{R} \{0,1\}^l; \\ m \xleftarrow{R} \{m_0, m_1\}; c \xleftarrow{R} \mathcal{E}(e,m)\Big] < \frac{1}{2} + \frac{1}{Q(k)}$$

Now, we want this to be an incremental scheme, and as it stands, it does not function as one. So, as before, we have to take this model, and add one more requirement that lets us make it incremental:

Definition 5.3 *An* incremental public-key encryption scheme *is a quadruple* $(\mathcal{G}, \mathcal{E}, \mathcal{D}, \mathcal{I})$ *of probabilistic polynomial time Turing machines, together with an indication of key length* k*, satisfying the conditions set in definition 5.1 (the first three listed), plus one more dealing with incrementation (the last one):*

key generation algorithm: *a probabilistic expected polynomial time algorithm* $\mathcal{G}$*, which on input* 1^k *produces a pair* (e,d) *where* e *is the public key and* d *is the private key:* $(e,d) \in \mathcal{G}(1^k)$.

encryption algorithm: *a probabilistic polynomial time algorithm* $\mathcal{E}$ *which takes as input a security parameter* 1^k*, the public key* e *from* $\mathcal{G}(1^k)$ *and a string* $m \in \{0,1\}^k$*, and outputs* $c \in \{0,1\}^*$ *which is the encryption of* m *using* e *as the key.*

[2]definition from [4]

decryption algorithm: *a probabilistic polynomial time algorithm $\mathcal{D}$ that takes as input a security parameter 1^k, a private key d from $\mathcal{G}(1^k)$, and a ciphertext c from $\mathcal{E}(1^k, e, m)$, and produces m.*

incrementation algorithm : *a probabilistic polynomial-time algorithm $\mathcal{I}$ which takes as input a security parameter 1^k, the public key e from $\mathcal{G}(1^k)$, a previous encryption of a message $c_0 = \mathcal{E}(1^k, e, m_0)$, and a list of changes to the message $m_0 \Rightarrow m_1$. It produces a new encryption $c_1 = \mathcal{I}(1^k, e, c_0, (m_0 \Rightarrow m_1))$ s.t. $m_1 = \mathcal{D}(1^k, d, c_1)$.*

And as usual, extend the security definition to cover this situation:

Definition 5.4 *Incremental polynomial-time indistinguishable security holds for an incremental public-key scheme if the following hold:*

- *Polynomial-time indistinguishable security holds when the scheme is used non-incrementally.*

- *An adversary cannot distinguish between any pair of: encryption formed via the encryption algorithm, and an encryption formed after using the incrementation algorithm on a previous encryption.*

  Let $\Pi(S)$ be a random permutation of the set S. For every PPT A, for every message length l, and for every polynomial Q, $\exists k_0$ s.t. $\forall k > k_0$:

$$\begin{aligned} Pr\Big[A(1^k, e, \{c_0, \dots, c_j\}) = b \quad | \quad & (e,d) \xleftarrow{R} \mathcal{G}(1^k); \{m_0, m_1, \cdots, m_j\} \xleftarrow{R} M(1^k); \\ & b \xleftarrow{R} \{0, \dots, j\}; \pi \longleftarrow \Pi(\{i = 0, \dots, j \quad | \quad i \neq b\}) \\ & c_b = \mathcal{E}(1^k, e, m_0) \\ & \forall i \in \{0, \dots, j-1\} : c_{\pi_i} = \mathcal{I}\big(1^k, e, c_{\pi_{i-1}}, (m_{i-1} \Rightarrow m_i)\big) \\ & \quad \Big] < \frac{1}{2} + \frac{1}{Q(k)} \end{aligned}$$

  where the probability is taken over the choice of $e, d, b, m_i's$, the permutation, and the random inputs. The encoding scheme for $\Rightarrow$ can be any scheme desired

as long as it holds the property that given m_1, and $(m_1 \Rightarrow m_2)$, anyone can compute the value of m_2.

That is, we want the scheme when used without incrementing to be as secure as non-incremental schemes. Furthermore, we want to guarantee that if we increment, the adversary will learn nothing about what was incremented, or be able to distinguish an incremented ciphertext. We might also want to add the following requirement: that given an incrementation to an encryption, the adversary learns nothing of how the source document changed. Simply stated, given $C_0 = \mathcal{E}(1^k, e, m_0)$ and $C_1 = \mathcal{I}\big(1^k, e, c_0, (m_0 \Rightarrow m_1)\big)$, the adversary knows nothing of $(m_0 \Rightarrow m_1)$. It can't even guess where a bit has changed. This requirement is not part of the security definition since it is both hard to achieve, and is usually not necessary in most schemes. However, it is a problem worth studying.

5.2 A Public-key Encryption Scheme

The approach suggested in chapter 6 does not work here for the simple reason that while it is easy to generate a pad with the private key, having the public key does not help re-create this pad which is necessary to XOR out of the encryption. One solution to this is to encrypt a randomly generated session key using a public-key scheme, and then increment with a symmetric scheme. Since this relies on two different style-encryption scheme, it will be analyzed later, but we will propose yet another approach.

Our approach is to simply divide the document into blocks, and encrypt each block separately. Then, the recipient decrypts each block and concatenates them to form the plaintext. However, if this scheme is used (as most public-key schemes are sometimes) in reverse, with the private key being used to encrypt and the public-key to decrypt, then this scheme becomes weak. Notably, an adversary can move, duplicate, remove, and substitute blocks in an undetectable manner. Therefore, we opt for a slightly slower scheme which more closely resembles other commonly used public-key encryption schemes (such as RSA). However, if encryption is the only

feature this scheme is being used for, then the just-mentioned scheme will suffice.

The scheme which we suggest uses the concept of blocks chained together by double-pairing a block with its predecessor and successor. This ensures a chain, while allowing insertions in the middle without renumbering tags. There is a worry that two blocks will end up being the same, allowing replacements of the chain from those particular points. To prevent this, a security randomizer is inserted to lower the probability of this happening. Furthermore, the encryption disguises when these events actually occur, so the probability of an adversary noticing this is extremely small.

There is one thing to note about the scheme proposed: It requires that temporary data be preserved in order to facilitate the computations, and that this temporary data is never made available to the adversary. This goes against the description of the incrementation algorithm (which only takes the encryption, and no internal state data), but is easily workable. While this requirement is usually not a problem, (and indeed prevents other people from incrementing on your own data unless you permit them by revealing this temporary data), this may not always be achievable. After the description of the scheme, an alternative approach which solves this problem will be explained, but not analyzed in depth.

5.2.1 The underlying algorithms

We describe how to form an encryption of a document using the four basic operations: **CREATE()**, **INSERT()**, **DELETE()**, and **MODIFY()**.

Create

The following is how an encryption of a document is performed (see figure 5-1). This is the **CREATE()** operation, which takes as input the document to be encrypted P, and yields an encryption of it C, and the structure necessary for incrementing S. S encodes the T's and C's (we can't increment only based on C, since if our scheme allowed that, than anyone could tamper with our encryption).

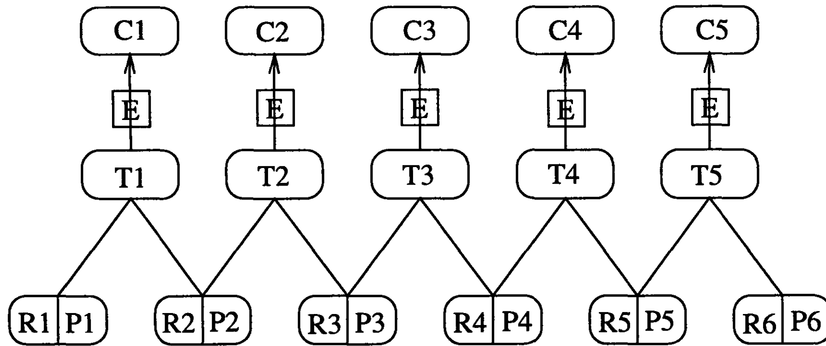

Figure 5-1: Incremental Public Key Encryption Description

- The document is divided into blocks of constant size (P_j)
- Each block is then appended or somehow combined with a randomizer of size k, where k is the security parameter (R_j).
- a temporary block to encrypt is then formed from respective pairs of blocks ($T_j = R_j \cdot P_j$).
- Each of these T_j's is encrypted using a standard public-key encryption scheme to yield a chain of ciphertexts $C_j = E_{pk}(T_j)$. Those ciphertexts are then combined to form the final ciphertext, which is the encryption. The encoding necessary for incrementation is the T_j's.

This ciphertext is then transmitted in the standard way. Then, we can perform incremental operations on S.

Insert

The first, and probably most common modification, is the **INSERT**(S, i, b) operation. S is the structure to be modified, i is the position where the insertion is to take place ($i = 0$ means insert at the beginning), and b is the block to insert. The insertion is performed in the following manner:

1. If $i = 0$, then we are prepending. This operation is simple:

   (a) from T_1 get $R_1 \cdot P_1$.

(b) generate R_0, and let $T_0 = (R_0 \cdot b) \cdot (R_1 \cdot T_1)$.

(c) let $C_0 = E_{pk}(T_0)$.

The same concept can be applied to appending.

2. If we are inserting in the middle, then the following is done:

   (a) from T_i get $R_i \cdot P_i$, and $R_{i+1} \cdot P_{i+1}$.

   (b) shift the reference numbers, so that $\forall j > i, T_{j+1} = T_j$ (so we can replace T_i with two blocks).

   (c) generate a new randomizer R, and let $T_i = (R_i \cdot P_i) \cdot (R \cdot b)$. let $T_{i+1} = (R \cdot b) \cdot (R_{i+1} \cdot P_{i+1})$

   (d) recompute $C_i = E_{pk}(T_i)$ and $C_{i+1} = E_{pk}(T_{i+1})$.

   (e) (reorganize the numbers to accommodate the insertion).

This operation can be amortized when several contiguous blocks are inserted.

Delete

We also need to handle a **DELETE**(S, i) operation. S is the structure, and i is the position of the block to be deleted.

1. If $i = 1$ or $i = n$, then we are deleting the first or last block. Simply remove T_1/C_1 or remove T_{i-1}/C_{i-1} respectively.

2. else we're removing in the middle, so we need to recompute some things:

   (a) delete T_i and C_i.

   (b) from T_{i-1} get $R_{i-1} \cdot P_{i-1}$, and from T_{i+1} get $R_{i+1} \cdot P_{i+1}$. Let $T_{i-1} = (R_{i-1} \cdot P_{i-1}) \cdot (R_{i+1} \cdot P_{i+1})$.

   (c) set $C_{i-1} = E_{pk}(T_{i-1})$

   (d) (reorganize the numbers to match the new layout).

Like with the **INSERT()** operation, **DELETE()** can be amortized when several contiguous blocks are deleted.

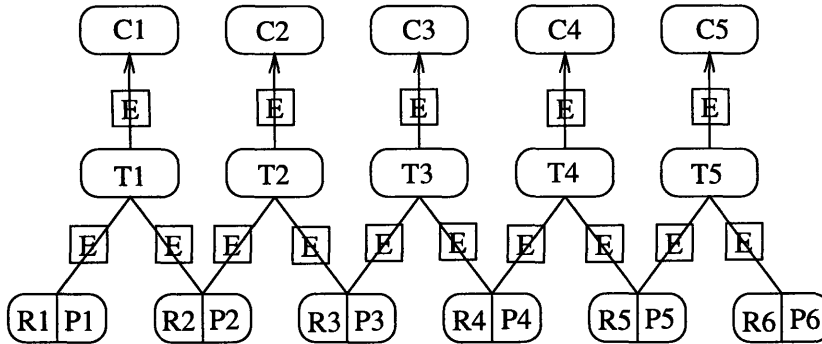

Figure 5-2: Appropriate Incremental PKE Description

Modify

Finally, the **MODIFY**$(S, i, (old \Rightarrow new))$ operation is the easiest to perform. S is the same structure to modify, i is the position where the modification will occur, and $(old \Rightarrow new)$ are the instructions on how to modify the block (probably just new contents for the block, but this is up to implementation).

1. generate a new R_i and replace P_i.
2. recompute the values of T_{i-1} and T_i where possible (if it's the first or last block, only one of those will exist).

5.2.2 The appropriately incremental algorithm

As was mentioned earlier, the above-mentioned scheme requires intermediate state (notably the values of T_i) in order to increment. This is not always acceptable, and so a simple solution to this problem is to also encrypt each $(R_i|P_i)$ element before pairing and merging them into T_i (see figure 5-2). This of course means that the incrementor can no longer actually know what is in the document, but he can increment successfully using the methods described above (encrypting in the obvious places). An immediately obvious disadvantage of this scheme is that it requires more encryptions and decryptions, which makes it slower. It otherwise is comparable to the previous scheme, so the following analysis applies to it also. However, since the temporary data is usually available, the previous scheme is a better idea.

5.3 Analysis of Scheme

Now that the scheme is described, it needs to be analyzed and compared to other non-incremental schemes to argue for its worth.

5.3.1 Efficiency

The biggest reason for using incremental cryptography is the efficiency gains made in computing the encryptions. As such, this scheme needs to appear more efficient than other schemes.

Time

To create an encryption from scratch, or to verify, one needs to compute for each of the blocks using the underlying encryption scheme n times. Standard schemes do one of two things: either encrypt a session key for a fast encryption scheme such as IDEA using the public-key cryptosystem, or else break the document up into manageable blocks and encrypt those. The first is VERY fast, much faster than the incremental scheme, but read the later section on security for an analysis of why this could be bad. The latter is still somewhat faster, since there is less data to encrypt, but now we have reached a factor of less than two, and this is on creation only.

Incrementing is very fast relative to the second scheme, since only two encryptions need be computed (at most), and not the entire document as is the case for the non-incremental scheme. Furthermore, the size of a block is variable, so it can be optimized as well as any other blocking scheme. This scheme ends up being fast when used mostly for updating encryptions.

Space

Space considerations are another issue. The incremental scheme is less efficient than other schemes in terms of space, but it is still on the same order of growth as others. The first nonincremental scheme described takes a little extra space to store the session key, but is otherwise the same length as the document. The second scheme

keeps completely to the length of the document. The incremental scheme, however, functionally doubles the length of the document and then encrypts in blocks.

From a mathematical standpoint, let

5.3.2 Security

The scheme needs to be compared to other commonly used schemes.

- The first scheme to compare to works the following way:

  - Create a random session secret key s to use in some secret-key encryption scheme (say IDEA).
  - Encrypt s using the public key in the chosen algorithm.
  - Encrypt the document using s in a secret-key encryption scheme.

  This scheme has both advantages and disadvantages:

  - Advantages:

    1. Since secret-key schemes today are *much* faster than public-key schemes, there is a general tendency to prefer them for encrypting large documents. This scheme is several orders of magnitude faster in encryption and decryption than using a public-key system entirely, and so is often used in systems that try to offer public-key encryptions (such as PGP). For the same reason, decryption is always faster than our system, and encryption is faster when an entire document is being encrypted (instead of dynamically updating an encryption).
    2. The same document can be encrypted to multiple recipients without re-encrypting for each user. Simply encrypt the session key for every user, and then send the same encrypted document plus the appropriate encryption of the session key.

  - Disadvantages:

1. Since this system is based on two cryptosystems, it has two possible points of failure: breaking either cryptosystem renders this scheme useless.
2. Secret-key encryption systems are not usually based on mathematical principles, so it is impossible to provide security guarantees for this cryptosystem (such as: "Breaking this is as hard as factoring").
3. This system relies on reliable pseudo-random number generation. This has turned out to be a big problem for systems-designers, and has led to many flawed implementations. Notably, the random number is not critical in our system, but predicting it in this system is enough to break it.

- Another system avoids using a secret-key system, but at cost. It works by simply taking the data, and encrypting it wholesale (some breaking up of the document into blocks may be necessary). This system also has advantages and disadvantages over our system:

  – The advantages include:

    1. This scheme is somewhat faster than our scheme, since it basically encrypts as few times as possible using the public-key scheme. This is slower than the above scheme, but faster than our scheme unless we also use blocks that are as big as possible.
    2. Being a very simple scheme, it is easier to prove things about this scheme.

  – The disadvantages include:

    1. It is not incremental, so minor changes still lead to complex computations, and it is impossible to design a system that will dynamically generate an encryption for the user.
    2. when encrypting for multiple recipients, since everything goes at the end and their is no standard optimization for this scheme, it takes

even longer to finish encrypting.

One thing to note about this scheme is that it is fully reliant on the public-key scheme, which makes it simpler to prove certain properties. For example, If we choose to use Rabin for our public-key encryption underlying scheme, we can demonstrate that the scheme is as hard as factoring.

5.3.3 Privacy

Ignoring the random numbers used to pad the blocks, this data structure encodes only one possible way to encode any document. Since the random numbers can't be predicted, there is no specific pattern to them. Therefore, it should be impossible to tell when and how blocks were inserted. This scheme therefore assures perfect privacy.

There is a worry that the underlying block scheme may leak some information, especially since it is known that there is a small relationship between subsequent blocks in the ciphertext. The security of that depends on the underlying public-key scheme. It was demonstrated in [1] specific bits within RSA and Rabin were 'secure', but formal analysis of this kind of usage of specific ciphers has not been performed. Nevertheless, it is highly likely that this usage of a cipher will not significantly weaken it.

5.4 Implementation

An incremental system was implemented and tested on a Sparc5 running Solaris 2.3. Four separate modules were written, which together interact to create the PKE mode for Emacs. Some of the code used **RSAREF** from RSA labs to achieve the underlying public-key encryptions. These modules were then used and evaluated to determine their efficiency. Since it was only meant for forward encryption, the chaining concept was not added, but a simple doubling of most timing computations are all it takes to convert for worst case.

5.4.1 The modules

Four specific modules were created which work together in the following way:

PKE.el is the main module. It is elisp code to create a minor mode in emacs which notices changes to the buffer and deals appropriately. The code is available in section A.3.1. It is activated by typing `M-x PKE-mode` within a buffer, or loading a file for which an old encryption has already been generated.

PKE.c is the code that **PKE.el** communicates with. It receives information following the protocol described in section 5.4.2 and creates the encryption with it. It is also responsible for saving the encryption and loading old encryptions when appropriate. Code is in section A.3.2.

PKE-keygen.c is responsible for generating key pairs for users whose names are supplied on the commandline thus: `PKE-keygen {username}`. It puts the keypairs in a private directory in the creator's path, where **PKE.c** and **PKE-decrypt.c** can use it. Code is in section A.3.3. To use it, simply run `PKE-keygen {username}` .

PKE-decrypt.c is the final component. It non-incrementally decrypts the document. Source for it lies in section A.3.4. To use it, run `PKE-decrypt {username} {infile} {outfile}` .

5.4.2 API for Inter-Procces Communication

Breaking the mode up into modules allows different public-key encryption schemes to be used as the underlying scheme. All that is required is that the modules communicate using the following protocol described in table 5.1.

Once we have modules that communicate using this protocol, we can then analyze them for user-observed efficiency.

5.4.3 Analysis of Implementation

The scheme runs synchronously to the editor, and as such, the only possible timing issue is the time from when the user wishes to save the file to the point when the checksum is computed. This of course varies according to the speed at which edits were being made, but unlike with some schemes, the position of the edit is unimportant.

Timing Performance

This implementation uses the RSAREF toolkit (see [15]) which is not as optimized as it could be. Nevertheless, the incremental encryption works quickly enough that most editing in a buffer is updated instantly.

However, decryption is another story. Since decryption is performed at the time of the request, non-incrementally, the user has to wait for $O(n)$ decryptions. On a sparcl, this translated to about one character decrypted per second, which is very slow. If decryption is a common enough operation to warrant speeding it up, then the session key approach might be a better choice, even though it is far less secure.

However, there is another way to balance out the decryption/encryption rate. Since the size of the public exponent is directly related to the time it takes to encrypt a block, and the size of the private exponent is directly related to the time to decrypt, and since the larger the public exponent, the smaller the private exponent, it is possible to choose values for them that achieve the best tradeoff between the two.

Space Analysis

On average, an encrypted file using this scheme, with a 512-bit RSA key, is about 70 times bigger. The larger the key, the bigger the encrypted file. This works strongly against the scheme, but in practice, storage space and transmission rates are cheap and fast, so this is not a major concern in a most practical systems.

If this is a problem, it is possible to merge several blocks together, although this leads to the same problem as with other such schemes, where one insertion in the

beginning leads to a recomputation everywhere else.

Command	Description
Public-key: {key value – encoded string}	This sets the value of the public key being used for encrypting the file. If there is any data, it is lost.
Create: {byte 1 – string} {byte 2 – string} : End	This creates a new structure, potentially dumping an older existing one. The bytes are encoded in decimal ASCII.
Insert: {position – integer} {data – encoded string}	This inserts *data* into the structure at position position. 1 makes it the first block in the structure.
Modify: {position – integer} {data – encoded string}	This changes the value at *position* to *data.*
Delete: {position – integer}	This removes block number position. 1 is the first block.
Save: {filename – string}	This saves the encryption to the filename. It is recommended that filename be an absolute pathname. returns: Done when done.
Load: {filename – string}	This loads up the data stored in *filename.* The key should have already been set by this point, since setting it later undumps all the data.
Quit:	This tells the program it is time to quit. **NOTE:** this does not bother saving the data!! make sure you save first!

Every message is composed of a header identifying the message content values, a carriage return, and then the body expected from the program, followed by a carriage return. Responses are terminated by a carriage return. Unless stated otherwise, encoding is converting the ASCII char into its hex value representation.

Table 5.1: PKE-API specifications

Chapter 6

Secret Key Cryptography

This chapter will deal with applying incremental cryptography to secret key cryptography. It will suggest a symmetric block cipher scheme, and analyze it in terms of efficiency and in terms of security.

6.1 Symmetric Encryption Schemes

A symmetric encryption scheme (also known as a private key scheme) is defined thus[1]:

Definition 6.1 *An encryption scheme is a pair $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ of probabilistic, polynomial time algorithms together with a security parameter 1^k. The first algorithm is the* key generator, *which is a PPT algorithm that yields a secret key $K = G(1^k)$. The second algorithm is the encryption algorithm, which takes as input the key, and a message, and yields an encryption of it $C = E(K, M) \equiv E_K(M)$. The last is the decryption algorithm, which takes as input the secret key and an encryption, and yields the original plaintext message $M = D(K, C) \equiv D_K(C)$. We require that $\forall (C, M) \in \Sigma^*$, if $C = \mathcal{E}_K(M)$, then $M = \mathcal{D}_K(C)$.*

Furthermore, we want to set a security parameter on this scheme so that we can consider it secure. We use the indistinguishability definition, which [10] has shown to be equivalent to the semantic definition.

[1]definition from [4]

Definition 6.2 *Informally,* indistinguishable security *is achieved when an adversary, given a ciphertext C and two messages M_0, M_1, can't tell which of the two was encrypted to yield C. Formally:*

$\forall M$ polynomial time probabilistic Turing machines, $\forall (M_0, M_1) \in \Sigma^l$ possible documents of length l, b be a random bit uniformly selected from $\{0,1\},$, and let $C = E_K(M_b)$. We require that $\forall Q()$ polynomial functions, $\exists k_0$ s.t. $\forall k > k_0$:

$$Pr[M(M_0, M_1, C) = b] < \frac{1}{2} + \frac{1}{Q(k)}$$

where this probability is taken over the set of possible messages, the choice of keys, and the choice of b.

Now, we take the definition of a symmetric encryption scheme, and modify it to create an incremental symmetric encryption scheme:

Definition 6.3 *An incremental symmetric encryption scheme is a quad $(\mathcal{G}, \mathcal{E}, \mathcal{D}, \mathcal{I})$ of probabilistic, polynomial-time algorithms with these properties:*

Key generator G() : *takes as input a security parameter 1^k, and yields the secret key $K = G(1^k)$.*

Encryption algorithm E() : *takes as input a key K, and a message M. It outputs an encryption of this $C = E(K, M) \equiv E_K(M)$.*

Decryption algorithm D() : *takes as input a ciphertext C, and outputs a decryption of it $M = D(K, C) \equiv D_K(C)$.*

Incrementation algorithm I() : *takes as input a key K, a ciphertext C_{old}, and instructions on how to change the document (old $\Rightarrow$ new). It yields a new ciphertext $C_{new} = I(C_{old}, K, (old \Rightarrow new)) \equiv I_K(C_{old}, (old \Rightarrow new))$.*

We require the following properties:

- $\forall (C, M) \in \Sigma^*$, *if* $C = \mathcal{E}(M)$, *then* $M = \mathcal{D}(C)$

- $\forall (C_0, C_1, M_0, M_1) \in \Sigma^*$, *if* $C_0 = \mathcal{E}(M_0)$, *and* $C_1 = \mathcal{I}(C_0, (M_0 \Rightarrow M_1))$ *then* $M_0 = \mathcal{D}(C_0)$ *and* $M_1 = \mathcal{D}(C_1)$.

Now we have an incremental symmetric encryption scheme. BUT our definition of security still doesn't take the incrementality into account, so we change it somewhat now to do just that:

Definition 6.4 *Incremental indistinguishable security holds for an incremental scheme if two properties hold:*

- *The scheme, when used nonincrementally, provides indistinguishable security.*
- *When an adversary, given* $C_0 = \mathcal{E}(M_0)$ *and* $C_1 = \mathcal{I}(C_0, (M_0 \Rightarrow M_1))$, *as well as* (M_0, M_1) *he can't distinguish which* M_i *was responsible for which* C_i *for all possible messages.*

One other possible addition which would increase the security of the cipher is that given $C_0 = \mathcal{E}(M_0)$ and $C_1 = \mathcal{I}(C_0, (M_0 \Rightarrow M_1))$, we learn nothing of $(M_0 \Rightarrow M_1)$, (i.e. we don't know which bits changed). This is a very strict requirement, which is not usually needed. However, should this be desired, approaches which use oblivious RAM (see [13]) can be used (or some other technique).

6.2 An Incremental Block Cipher Symmetric Scheme

An efficient initial approach as proposed in [2] is to preserve an encrypted list of changes E_2 to the original document D (whose encryption is E_1). Then, whenever the encryption is requested, E_1 and E_2 are provided. When E_2 grows longer than some variable l, we integrate those changes back into D and recompute E_1, setting E_2 to an empty list. This can be pipelined, and the analysis of this scheme is done in that paper.

The real problem with the above technique is the lack of privacy for the incremental document, and so we propose another method for obtaining an incremental block cipher secret key cryptosystem.

This method can work with any block cipher. It allows for extremely efficient modify operations, and also allows for inserts and deletes, which, depending on the location, can be extremely efficient, or at the worse as costly as simply recomputing the entire encryption from scratch.

6.2.1 Description of Scheme

Flawed Approach

One possible way to have an incremental scheme is to use the block cipher in electronic codebook mode (ECB). This scheme divides the document up into blocks which can be moved around, as well as modified independently from the rest of the document. Unfortunately, this flexibility also makes the scheme very susceptible to substitution attacks (replacing one block with another and such). It is also susceptible to some types of analysis (for example, noticing a long sequence of zeros).

Another approach is to use the block cipher in output feedback mode (OFB). This again has the distinct advantage of dividing the document into blocks which are separately computed for, although the hash chain has to be precomputed. While one can't easily attack this scheme with substitutions and analysis, having just one plaintext/ciphertext pair is enough to break the entire scheme, easily producing all other possible pairs (though not knowing the key used).

We therefore want to design an approach that takes the best of both, and hopefully use their features to protect against the flaws.

Correct Approach

The following is how one generates an encryption $C = (IV, \{C_1 \cdot C_2 \cdot \ldots \cdot C_{n-1} \cdot C_n\})$ of a plaintext $P = \{P_1 \cdot P_2 \cdot \ldots \cdot P_{n-1} \cdot P_n\}$ where each $|P_j|$ is the size of a block in the cryptosystem being used (64 bits in DES). It requires a random initialization vector IV and a key k.

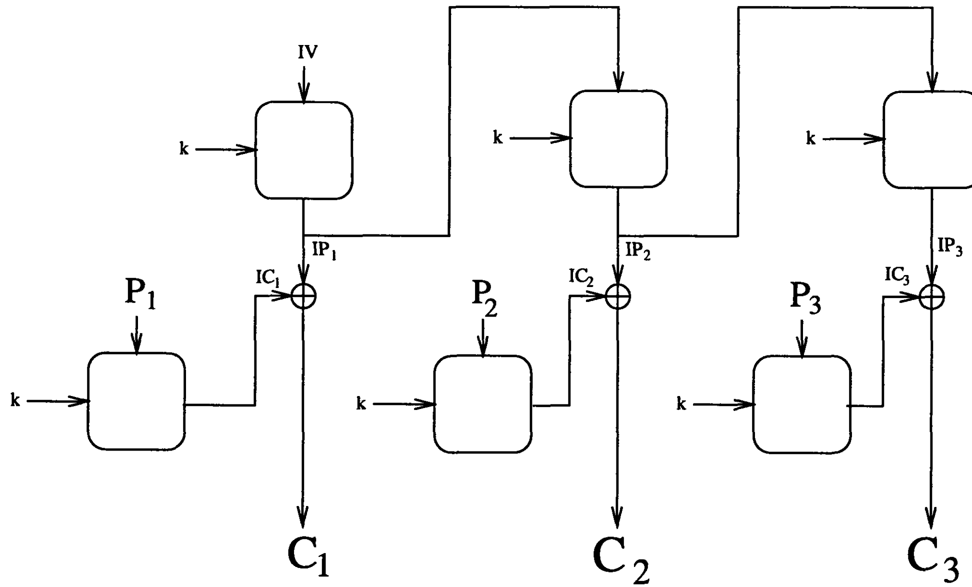

Figure 6-1: Incremental OFB-ECB mode for a block cipher

encryption

To encrypt P, perform the following steps (observe figure 6-1):

1. Create the intermediate pad IP in the following manner:

$$\begin{aligned} IP_1 &= E_k(IV) \\ IP_2 &= E_k(IP_1) \\ &\vdots \\ IP_n &= E_k(IP_{n-1}) \end{aligned}$$

2. Create the intermediate block cipher using ECB mode on the plaintext:

$$
\begin{aligned}
IC_1 &= E_k(P_1) \\
IC_2 &= E_k(P_2) \\
&\vdots \\
IC_n &= E_k(P_n)
\end{aligned}
$$

3. Now put the two together using XOR's:

$$
\begin{aligned}
C_1 &= IP_1 \oplus IC_1 \\
C_2 &= IP_2 \oplus IC_2 \\
&\vdots \\
C_n &= IP_n \oplus IC_n
\end{aligned}
$$

The above technique should yield the desired ciphertext.

decryption

In order to decrypt the above generated text, one needs the IV, the C, and the k. Then, it's just a matter of running things in reverse (observe figure 6-1):

1. Create the intermediate pad IP as before:

$$
\begin{aligned}
IP_1 &= E_k(IV) \\
IP_2 &= E_k(IP_1) \\
&\vdots \\
IP_n &= E_k(IP_{n-1})
\end{aligned}
$$

2. Create the intermediate ciphertext IC in the following manner:

$$\begin{aligned} IC_1 &= D_k(C_1) \\ IC_2 &= D_k(C_2) \\ &\vdots \\ IC_n &= D_k(C_n) \end{aligned}$$

3. Finally, extract the plaintext thus:

$$\begin{aligned} P_1 &= IP_1 \oplus IC_1 \\ P_2 &= IP_2 \oplus IC_2 \\ &\vdots \\ P_n &= IP_n \oplus IC_n \end{aligned}$$

incrementing

An operation which modifies data in a block requires that entire block be recomputed. Efficiently done, this requires the new plaintext NP be encrypted, as well as the old plaintext OP. The new ciphertext block will look like this:

$$C_n \longleftarrow C_n \oplus E_k(OP) \oplus E_k(NP)$$

Operations which add or remove data work in four parts, and this is the high-level overview of what needs to be done (instructions here are for removal, but easily map to addition):

- First, remove entire blocks that need to be removed. This only happens if the delete is bigger than the size of a block AND happens to erase across boundaries.
- keep track of the values of all the IC's, since some will have to be shifted forward.

- Then, shift all the data ahead down. This will look like a modify operation, done on all future blocks.

- Recompute all the values that have changed.

As can be seen, in the worst case, a deletion of less than a whole block from the beginning can set off an entire re-computation. It is possible to create special characters to represent holes to avoid this, but this will create a history, which is something we wish to avoid.

6.2.2 Analysis

This scheme is analyzed in terms of security, privacy, and efficiency.

security

In order to analyze this scheme, we will assume that the block cipher used is a perfect pseudorandom permutation (as done in [6]). However, even if it is not, it seems likely that any attack performable on OFB mode will apply here, as well as cryptanalysis on ECB mode.

First we need to have a way of talking about the secret-key encryption scheme, and so we regard it as a pseudo-random permutation. But to define a pseudo random permutation, we first need to explain a distinguisher. Intuitively, a distinguisher is a function that attempts to tell whether some "random" string it is viewing is the output of a pseudo-random function or truly random data, with a limited amount of resources, but with access to the pseudo-random function.

Definition 6.5 *given a family of functions F, the* advantage *of a distinguisher function is $Adv_D(F) = |P_D(F) - P_D(R)|$ where:*

$$
\begin{aligned}
P_D(F) &= Pr[D^g = 1 : g \xleftarrow{R} F] \\
P_D(R) &= Pr[D^g = 1 : g \xleftarrow{R} R]
\end{aligned}
$$

Definition 6.6 *F is a (t, q, ϵ)-*secure finite pseudorandom function family *if any distinguisher D who makes at most q queries and runs in time at most t has $Adv_D(F) \leq \epsilon$.*

Informally, we want it to be hard to predict the random numbers, and we can make it as hard as we want by controlling the variables.

Claim 6.1 *Assuming the block cipher $f_k()$ is chosen from a family of pseudorandom functions where k is secretly agreed upon. This encryption mode,* WITHOUT *the incrementation algorithm, yields a cryptosystem that offers indistinguishable security if ECB and OFB modes for the block cipher do.*

Sketch of Proof (informal argument) The intermediate pad produced (IP) is the same pad as produced by the standard OFB mode for a block cipher. The only functional difference is the fact that instead of XORing directly with the plaintext, we now XOR with the plaintext encrypted in ECB mode.

Any given ciphertext block does not affect the value of the next one (although there is a relationship based on the chaining). Therefore, given the assumption that OFB mode is secure, the only way the scheme could be made weak in this system is through the encryption of the plaintext block before the xor. Given that the blocks are independent at that point, one need only consider the effect on a per-block basis.

We know that the adversary can't distinguish between $IC_j \oplus P_j$ and $IC_j \oplus random_data$, so if he can distinguish between $IC_j \oplus E_k(P_j)$ and $IC_j \oplus E_k(random_data)$, there is something inherent to the encryption that gives away this information. BUT since the cipher scheme is supposed to be a pseudorandom permutation, this can't

be so. Therefore, assuming the OFB mode is secure, and that the block encryption scheme used is secure, then so is this scheme. ■

Although the scheme is at least as secure as OFB when used once, the fact that is being used incrementally opens up a different venue of attack. The adversary can attempt to look at multiple revisions of the document in the hopes of gleaning extra information.

Claim 6.2 *The adversary, when allowed to see updates to the encryption, can do no better with a chosen plaintext attack (where he may query both an oracle for $\mathcal{E}$ and an oracle for $\mathcal{I}$) than having a chosen plaintext attack on the standard block cipher.*

Sketch of Proof (informal) The basic idea is to demonstrate that any chosen plaintext attack can be done by performing a chosen plaintext attack on the block cipher itself. (i.e. one can simulate the attacks by performing similar subattacks on the block cipher itself).

Given the underlying block cipher algorithm as a black box, it is simple to choose a random number as IV and run it through the cipher chaining to get the pad. Then for any block, simply encrypt the message and XOR it in. This should yield plaintext/ciphertext pairs as desired.

Note that this is not demonstrable for a chosen ciphertext attack, unless one is restricted to IV being randomly selected for the chosen ciphertext. (It is impossible to derive the cipher chain from a chosen IV with a chosen ciphertext attack on the boxes, and so unless one can allow that to always be random and derive the chain backwards, this scheme may be weak to a wise chosen ciphertext attack with a specifically chosen IV). Although it has not been proven, the author suspects that this scheme is still strong against chosen ciphertext as long as the basic block scheme is not susceptible to a combination of chosen plaintext AND chosen ciphertext attacks. This is based on the fact that the choice of IV can give chosen plaintext information even during a chosen ciphertext attack. ■

We also note that if the underlying block cipher scheme offers indistinguishable

security, so does the entire scheme.

Finally, there is one more concern that needs to be addressed with this scheme: the cipher can be truncated to yield another valid ciphertext. Since we still do not have MACs to deal with this problem (those will be discussed in chapter 4), we can instead set the first or last blocks to be the length of the document. Being a block, they are easy to update when the document length changes, and being part of the encryption, they are theoretically hard to break (although one can easily derive one block's encryption from it).

privacy

This scheme encodes no history, so it can be considered private, assuming that the above security holds (in this sense, our data structure is an oblivious one, since there is only one possible representation for any data). The privacy of the encryption however, is still fundamentally tied to the privacy of the block scheme. The OFB mode will provide some randomization of input to protect equivalent plaintexts from being immediately identifiable. However, guarantees can only be provided through the choice of block encryption scheme, not just by examining the usage of it.

efficiency

Not only is the above scheme secure, it is computationally slower than standard encryption (using CBC, OFB, or ECB mode) by only about a factor of two. This is for a computation of an encryption/decryption from scratch. As soon as incrementality is introduced, this scheme becomes much faster than the CBC or OFB mode.

As was mentioned earlier, however, this scheme can become extremely slow of the prevalent type of operation is an insertion or deletion of date that is not aligned on a block boundary and whose size is not a multiple of the size of a block. Therefore, it is recommended that the types of operations normally performed on the document be analyzed (potentially using the *usage analyzer* discussed in chapter 3) and appropriate variables tested. One can easily change the size of a block, pad, and amount of time waiting before attempting to compute for a block. And by setting those to appropriate

values, achieve optimal efficiency.

Chapter 7

Directions for Incremental Cryptography

The previous chapters analyzed several cryptographic primitives and how to make them incremental. This chapter will discuss other ideas that are related to the topic, and also suggest several possible applications of incremental cryptography.

7.1 Other Related Ideas

Incremental cryptography deals with regenerating valid transformations of modified documents. There are other similar ideas that come out of this concept, and merit some discussion.

7.1.1 Incremental Hashing

Another oft-used cryptographic primitive is the cryptographic hash. This primitive takes a long document and generates a fingerprint of it that has several properties. Two of the relied on properties include the fact that it is hard to guess any document with that fingerprint, and also that it is hard to find two documents with the same fingerprint. This primitive is used in many protocols to commit without revealing information, or to compute partial checksums.

As with other schemes, this can and should be made incremental so that it is possible to update these fingerprints without recomputing from scratch. Some work has been done towards this goal, but there are many properties of hashes, and so not all schemes fit all these properties. However, this is something that will be examined over time.

7.1.2 Incremental Digital Signatures

While a signature can be created by using the public key scheme mentioned earlier (reversing the keys used to encrypt and decrypt), it is nevertheless not as efficient as it can be given that a signature needs to be a small and hard to forge item, and not an encryption that can only be created by the writer. As such, research has been undertaken into exactly how big a signature need be in an incremental scheme, and also into designing some schemes (one scheme with interesting properties is suggested in [12]), and which uses an oblivious 2-3 tree to store the document in a manner which disguises how the document arrived at its current state. However, it has a few undesired features including a long signature, and several complex computations need to be performed with every update. It is therefore likely to need some implementation tests and optimizations to see whether it can work fast enough to be inobtrusive to the users.

Another potential approach is to use the incremental hashing proposed in section 7.1.1, and then to encrypt it with the private key. However, since the encryption is likely to be the much more expensive operation, this may not be much of a win. However, there are schemes (such as ElGamal) which allow for some precomputation ahead of time, thereby leaving just a small amount of work at the end. However, since incremental hashing schemes weren't covered, it's hard to know how well this would do too.

7.1.3 Incremental Group Protocols

There are many protocols in the real world which require a set of participants. In a real-world application, this set may vary with time (an employee leaves, a new employee joins, etc). Currently, when this happens, most schemes require generating the respective components from scratch, with everyone meeting together. It should be possible instead to, given specific security criteria, deal without starting everything from scratch. This of course can lead to interesting problems with revoking people who have left, but nevertheless, is an interesting direction to examine.

7.1.4 Incremental Keys

While it is probably more likely that a document changes, it is also possible to have a system where the key can undergo changes instead of the document. One simple solution to this idea is to generate a session key, and transform the document using the session key. Then, encrypt the session key with the main key (the key that will undergo changes). Then, every time the main key changes, just re-encrypt the session key.

While this approach is simple, it doesn't always achieve what is desired from a cryptosystem, and as such, other directions may merit exploring. One such direction is explored in a paper by Blaze and Strauss (see [7]), where they suggest a proxy encryption function to later be replaced by another key.

One possible use for this technology is the case of encrypted mail to multiple recipients. It would be helpful if one could easily modify an encryption to one person into an encryption for another without recomputing from scratch.

7.2 Possible Uses

Apart from using incremental cryptography to dynamically generate signatures, encryptions, or MAC for buffers as they are being edited, incremental cryptography can be put to use in many other real-world applications.

7.2.1 Filesystems

One application of incremental cryptography is within filesystems, either as a form of virus protection, or for the protection of the data. Studies of filesystems such as UFS (see [14]) indicate patterns among specific paths. Some file tend to be temporary files created and deleted in their entirety (file in /tmp/ often exhibit this behavior for example), others (like logs) are appended to mostly. Finally, there are files which have appends, modifies and deletes performed on them. LFS takes advantage of this by making all changes on a "log" which they regularly prune. Instead, we can stick to the UFS semantics, but every time a modification/insertion/deletion occurs, we generate a MAC or an encryption of the sector or the file.

Using a MAC lets us notice when a file has changed without permission, and building it into the filesystem will make it automatic. Using encryption allows us to prevent unauthorized people from reading any data they were not approved for.

7.2.2 Blinding

Since there is nothing in the definition of an incremental scheme that requires the incrementor to know the original document, interesting protocols which require that the source document be changed in a particular way by a party which has no knowledge of the actual document can be designed. One such application is blind signing, where the signer can modify the final block of a document with some signature without actually seeing the rest of the document.

7.2.3 Web Pages

Many types of services on the web now want to implement encryption for data transmitted. While many things that are transmitted are static pages (for example, an online dictionary), others are not. Many sites have shopping carts that you can load with things, thereby adding an item at a time. Also, there are pages with fill-out forms. Both these things seem ideal for incremental encryption. In the first, every item that is added is functionally one *insert()* operation. On the other, the fill-out

form is usually a standard form which the web server may have pre-computed an encryption for a long time ago. Then, for filling it out and sending it back, a few *modify* or *insert* operations will fill-out the form and have it ready to be sent back encrypted.

7.2.4 Electronic Cash

There are many schemes proposed for conducting commerce on the Internet. Some are based on the concept of an electronic wallet that is carried around by the user, and which changes after every sale is performed. Digital signatures are used to make sure the user is authenticated, and also to make sure that the wallet is valid according to some bank. Since the wallet changes tend to be of the nature of "add fifty dollars" or "remove fifty cents", it is possible that incremental cryptography can be used here too.

Chapter 8

Conclusion

As was stated in the introduction, cryptography is not used much in real-world applications even in situations where it is an ideal solution to the problem. This problem has been one that cryptographers have worked hard at solving, and in order to do so, several problem areas were identified.

Incremental cryptography has been demonstrated to be very effective in reducing the user-apparent delays, as well as making cryptography more embedded within the applications. While it does nothing to solve other problems (such as political ones), it does provide a major step in the direction of a long-term solution to the problems of *ease of use*, *perceived speed*, and *ease of integration*.

As part of this thesis, several incremental cryptographic transformations were described and analyzed, and some were implemented to measure performance and to learn from using an actual system. The speed of the incremental algorithms was discovered to be efficient enough to allow for constant updating without worry in both implementations.

Finally, the implementations led to studying different approaches at optimizing the increment operations, and usage patterns were studied to help figure out values for specific variables within schemes.

Overall, in conclusion, incremental cryptography appears to be a very solution to these problems, and it is hoped that in the future, software programmers will integrate the concepts into their applications, and users will have this features without worrying

about how they work, how to use them, or about waiting for things to encrypt and decrypt.

Appendix A

Source Code

A.1 MAC-mode

The following is code used to implement a MAC minor mode in emacs. When turned on for a buffer, it will automatically attempt to compute the MAC for the particular buffer. Whenever the buffer is saved, the MAC is saved along with it in a file with the same name and the suffix of '`.sum`'.

There is documentation within the code to explain what is happening, but the basic idea is that MAC.el is loaded and used via `M-x MAC-mode`. Then, for each buffer, it will ask for keys, and create two subprocesses, one for the f1() and one for the f2(). These processes do the complex computation in C and communicate back with emacs to deliver the results. See the chapter on MAC computations (chapter 4) for an explanation of how it works.

It is implemented as a minor mode in emacs to allow for easy integration with other features of emacs, such as the mail reader, or cc-mode editing, or any other major or minor modes that are being used.

A.1.1 MAC.el

This is the main elisp code to achieve the minor mode.

```
;;;  Code to create a minor mode that will notice when the buffer is
;;;  edited, and appropriately modify a local MAC code based on the
;;;  given keys as is documented in 'Incremental Cryptography and
;;;  Application to Virus Protection'. by Mihir Bellare, Oded
;;;  Goldreich and Shafi Goldwasser
;;;
;;;  Written by Yoav Yerushalmi under the supervision of Prof. Shafi
;;;  Goldwasser.  August 8, 1996
;;;
;;;  Permission is granted to use and modify this code as desired, as
;;;  long as credit is given to all the above.

;;; -----------------------------------------------------------------
;;;                          MAC  Mode
;;;   An elisp minor mode to generate a Message Authentication Code as
;;;   a buffer is being updated.
;;;
;;;   To use, simply run MAC-mode on a buffer, and edit it. When
;;;   trying to save, the checksum will be written out automatically
;;;   to a file named {foo}.sum .
;;;
;;;   This is still NOT a complete product. Please understand many
;;;   things still will change and be optimized.
;;;
;;;   Also, being a minor mode, you can have checksums being computer
;;;   for multiple buffers simultaneously (this is untested, but the
;;;   code is in). In fact, it will smartly try and keep up with all
;;;   buffers in the background, updating checksums as they occur.
;;;
;;; -----------------------------------------------------------------
;;;  For the actual description of the scheme, please pick up a copy
;;;  of the paper
;;;
;;;  This is implementing the XOR scheme, and does so the following
;;;  way: a mac-list datastructure is created for the buffer. This is
;;;  basically a linked list, with the following properties:
;;;
;;; --------------    ------------    -------------    -------------
;;; | MAC-start- |    |   MAC    |    |   MAC     |    | MAC-end-  |
;;; |   block    |--->| datablock|--->| datablock |--->| block     |
;;; --------------    ------------    -------------    -------------
;;;
;;;  A MAC-datablock is composed of the following three things:
;;;      MAC-string  :  hopefully a MAC-blocksize length string
;;;                     representing the part of the buffer this block
```

```
;;;                     is holding.
;;;      MAC-pad      :  a randomly generated MAC-security-padding
;;;                      length string
;;;      MAC-sum      :  F1(cur-block + next-block). nil means
;;;                      uncomputed yet.
;;;
;;;  please note that under this setup, MAC-start-block's checksum
;;;  changes, while MAC-end-block's doesn't.
;;;
;;; -----------------------------------------------------------------

(require 'timer)

;;; Variables that can and SHOULD be changed depening on usage
;;; patterns
(defconst MAC-blocksize 64
  "number of bytes in a block, not including random padding")

(defconst MAC-security-padding 16 "number of bytes in the security
padding appended to a block -- security parameter")

(defconst MAC-idle-wait 0.5 "number of seconds to let the machine be
idle before trying to process blocks")

(defconst MAC-idle-repeat 0.3 "number of seconds between iterative calls when idle")

(defconst MAC-F1-proc "/mit/yoav/work/lcs/bin/MAC-f1"
  "the process to run to compute the f1 function")

(defconst MAC-F2-proc "/mit/yoav/work/lcs/bin/MAC-f2"
  "the process to run to compute the f2 function")

(random t)        ;;; initialize random seed using emacs shitty random
  ;;; numbers

(defvar MAC-last-idle-timer 'nil
  "a variable to hold the last idle-timer")

(defvar MAC-start-block (vector "start"
"startpad"
'nil)
  "first block in any MAC-list, sum changes and so block may change")

(defconst MAC-default-start-block (vector "start"
"startpad"
```

```
'nil)
  "default value of first block in any MAC-list")

(defconst MAC-end-block (vector "end"
      "endpad"
      'nil)
  "last block in any MAC-list")

(defvar MAC-buffers '()
  "A list of all buffers currently using MAC-mode")

(defvar MAC-f1-key 'nil
  "The variable used to hold the key for f1")
(defvar MAC-f2-key 'nil
  "The variable used to hold the key for f2")
(defvar MAC-f1-proc 'nil
  "The process that is used to compute MAC-f1")
(defvar MAC-f2-proc 'nil
  "The process that is used to compute MAC-f2")
(defvar MAC-f1-queue 'nil
  "The queue being used for encryption of blocks")
(defvar MAC-f2-queue 'nil
  "The queue being used for calculating final checksums")
(defvar MAC-mode 'nil
  "*Non-nil enables message authentication computations on buffer.
The default value is nil. To change the default, do this:
        (set-default 'MAC-mode t)")
(defvar MAC-mode-key 'nil
  "The secret key associated with the person doing the edits.
   hopefully the same throough all buffers")
(defvar MAC-local-h 'nil
  "The variable that holds the current XOR of all block checksums")
(defvar MAC-final-sum 'nil
  "The variable that holds the final encrypted checksum")
(defvar MAC-list 'nil
  "The list structure representing the entire buffer")
(defvar MAC-list-modified 't
  "This variable, when set to 't, implies the MAC-list datastructure
   may have shorter or longer strings than MAC-blocksize in them")
(defvar MAC-list-sums-complete 'nil
  "This variable, when set to 'nil, implies that if MAC-list-modified
   is 'nil, the sums are still not all correct")
(defvar MAC-list-final-sum-good 'nil
  "This variable, when set to 't, along with MAC-list-modified and
   MAC-list-sums-complete, decide whether the final checksum for the
```

```
   buffer is valid")

(defvar MAC-orig-require-final-newline 'nil
  "This variable holds the original value of require-final-newline")

(make-variable-buffer-local 'MAC-f1-key)
(make-variable-buffer-local 'MAC-f2-key)
(make-variable-buffer-local 'MAC-f1-proc)
(make-variable-buffer-local 'MAC-f2-proc)
(make-variable-buffer-local 'MAC-f1-queue)
(make-variable-buffer-local 'MAC-f2-queue)
(make-variable-buffer-local 'MAC-local-h)
(make-variable-buffer-local 'MAC-final-sum)
(make-variable-buffer-local 'MAC-start-block)
(make-variable-buffer-local 'MAC-mode)
(make-variable-buffer-local 'MAC-list)
(make-variable-buffer-local 'MAC-list-modified)
(make-variable-buffer-local 'MAC-list-sums-complete)
(make-variable-buffer-local 'MAC-list-final-sum-good)

(make-variable-buffer-local 'after-change-functions)

;  a few inline functions to obviate the MAC-list datastructure
(defsubst MAC-string (block)
  "return the string associated with the MAC-list structure block"
  (aref block 0)
)
(defsubst MAC-pad (block)
  "return the security pad associated with the MAC-list structure
   block"
  (aref block 1)
)
(defsubst MAC-sum (block)
  "return the sum associated with the MAC-list structure block"
  (aref block 2)
)
(defsubst MAC-set-string (block val)
  "set the value of string in block to val"
  (aset block 0 val))
(defsubst MAC-set-pad (block val)
  "set the value of pad in block to val"
  (aset block 1 val))
(defsubst MAC-set-sum (block val)
  "set the value of sum in block to val"
  (aset block 2 val))
```

```
(or (assq 'MAC-mode minor-mode-alist)
    (setq minor-mode-alist
  (cons '(MAC-mode " MAC") minor-mode-alist))

;; set to always check if there is any more updating to do.
;; this will keep checking constantly.
(run-with-idle-timer MAC-idle-wait
     'MAC-idle-repeat 'MAC-mode-idle)

(defun MAC-find-file-hook ()
   "hook to call after a file is loaded to
    see whether it has been saved with a MAC"
   (let* ((name buffer-file-truename)
  (sumname (concat name ".sum")))
     (if (file-readable-p sumname) (MAC-mode))))

(defun MAC-kill-buffer-hook ()
  "hook to call when a buffer is killed"
  (if (not (null MAC-f1-proc))
(delete-process MAC-f1-proc))
  (if (not (null MAC-f2-proc))
(delete-process MAC-f2-proc)))

(add-hook 'find-file-hooks 'MAC-find-file-hook)
(add-hook 'kill-buffer-hooks 'MAC-kill-buffer-hook)

(defun MAC-mode (&optional arg)
  "Minor mode for editing files with automatic MAC
   computation"
  (interactive "P")
  (setq MAC-mode
(if (null arg)
    (not MAC-mode)
  (> (prefix-numeric-value arg) 0)))
  (if MAC-mode
      (progn
(if (null MAC-buffers)
    (setq MAC-buffers (list (current-buffer)))
  (nconc MAC-buffers (list (current-buffer))))
(if (null MAC-f1-key)
    (progn
      (setq MAC-f1-key (MAC-get-f1-key))
      (let ((process-connection-type nil))
(setq MAC-f1-proc
```

```
      (start-process "F1"
     (generate-new-buffer-name " F1")
     MAC-F1-proc))
(if (null MAC-f1-proc)
    (error "Couldn't start up " MAC-F1-proc))
(process-kill-without-query MAC-f1-proc))
      (process-send-string MAC-f1-proc (concat MAC-f1-key "\n"))
      (setq MAC-f1-queue (tq-create MAC-f1-proc))))
(if (null MAC-f2-key)
    (progn
      (setq MAC-f2-key (MAC-get-f2-key))
      (let ((process-connection-type nil))
(setq MAC-f2-proc
      (start-process "F2"
     (generate-new-buffer-name " F2")
     MAC-F2-proc))
(if (null MAC-f2-proc)
    (error "Couldn't start up " MAC-F2-proc))
(process-kill-without-query MAC-f2-proc))
      (process-send-string MAC-f2-proc (concat MAC-f2-key "\n"))
      (setq MAC-f2-queue (tq-create MAC-f2-proc))))
(if (timerp MAC-last-idle-timer)
    (cancel-timer MAC-last-idle-timer)) ;;don't need the timer.
(add-hook 'after-save-hook 'MAC-save-checksum)
(setq after-change-functions
      (cons 'MAC-mode-buffer-change-hook after-change-functions))
(setq gc-cons-threshold 3000000)
(setq MAC-list (MAC-prepare-checksum))
(setq gc-cons-threshold 1000000))
    (setq MAC-buffers (delete (current-buffer) MAC-buffers))
    (setq MAC-list `())
    (delete 'MAC-mode-buffer-change-hook after-change-functions)
    (remove-hook 'after-save-hook 'MAC-save-checksum)))

(defvar MAC-mode-map nil "Keymap for MAC-mode.")

(if (null MAC-mode-map)
    (fset 'MAC-mode-map
  (setq MAC-mode-map (copy-keymap (current-global-map)))))

(if (not (assq 'MAC-mode minor-mode-map-alist))
    (setq minor-mode-map-alist
  (cons (cons 'MAC-mode MAC-mode-map)
minor-mode-map-alist)))
```

```
(defun MAC-mode-buffer-change-hook (beg end old-len)
  "This function is supposed to be called every time the buffer
   changes, and will contain information as to exactly HOW the
   buffer changed. This is then used to update the MAC-list
   structure"
  (if (timerp MAC-last-idle-timer)
      (cancel-timer MAC-last-idle-timer))
  (setq MAC-list-modified 't)
  (let ((prev MAC-list)
(block)
(new-string-start)
(new-string-end)
(new-string)
(pos beg))
    (setq pos (- pos (length (MAC-string (car (cdr prev))))))
    (while (and (> pos 1)
(not (eq (car (cdr prev)) MAC-end-block)))
      (setq prev (cdr prev))
      (setq pos (- pos (length (MAC-string (car (cdr prev)))))))
    (setq block (car (cdr prev)))
    (if (eq block MAC-end-block)
; we're at the last block, so
; we probably want to create
; an empty block to allow for
; things to insert into.
(progn
  (setcdr prev (cons (MAC-make-list-block "")
     (cdr prev)))
  (setq block (car (cdr prev)))
  (setq pos (+ pos (length (MAC-string MAC-end-block))))))
    (MAC-set-sum (car prev) 'nil)
    (MAC-set-sum block 'nil)
; pos points at exactly where
; we insert
    (setq pos (1- (+ pos (length (MAC-string block)))))
    (setq new-string-start (substring (MAC-string block) 0 pos))
; we're deleting some chunk
    (if (> old-len 0)
(progn
; we're deleting the first
; chars of the next block
  (if (= pos MAC-blocksize)
      (progn
(setq prev (cdr prev))
(setq block (car (cdr prev)))
```

```
(MAC-set-sum block 'nil)
(setq pos 0)
(setq new-string-start "")))
  (while (>= old-len (length (MAC-string block)))
    (setq old-len (- old-len (length (MAC-string block))))
    (setcdr prev (cdr (cdr prev)))
    (setq block (car (cdr prev))))
  (setq old-len (- old-len (- (length (MAC-string block)) pos)))
  (if (<= old-len 0)
      (setq new-string-end
    (substring (MAC-string block)
       (+ pos
  (+ (-
      (length (MAC-string block)) pos)
     old-len))))
    (setq new-string-end (substring (MAC-string block)
    old-len)))
  (setq new-string
(concat new-string-start
(concat (buffer-substring beg end)
new-string-end))))
      (setq new-string (concat new-string-start
       (concat (buffer-substring beg end)
       (substring
(MAC-string block) pos)))))

; new-string now holds the
; string that should replace
; the one in 'block' need to
; either just put it straight
; in, or else create multiple
; blocks for it.
    (if (< (length new-string) MAC-blocksize)
(if (= (length new-string) 0)
    (setcdr prev (cdr (cdr prev)))
  (MAC-set-string block new-string)
  (MAC-set-pad block (MAC-randomizer new-string)))
      (if (= (length new-string) MAC-blocksize)
  (progn
    (MAC-set-string block new-string)
    (MAC-set-pad block (MAC-randomizer new-string))
    (MAC-calculate-block-sum (car prev) block))
; our string is longer than a
; block, so put as much of it
; in the current block, and
```

```
; then insert the rest into a
; future block.
(MAC-set-string block (substring new-string 0 MAC-blocksize))
(MAC-set-pad block (MAC-randomizer (MAC-string block)))
(setq new-string (substring new-string MAC-blocksize))
(setq prev (cdr prev))
(while (>= (length new-string) MAC-blocksize)
  (setcdr prev (cons
(MAC-make-list-block
 (substring new-string 0 MAC-blocksize))
(cdr prev)))
  (MAC-set-sum (car prev) 'nil)

  (setq prev (cdr prev))
  (setq new-string (substring new-string MAC-blocksize)))
(if (/= (length new-string) 0)
    (progn
      (setcdr prev (cons (MAC-make-list-block new-string)
 (cdr prev)))
      (MAC-set-sum (car prev) 'nil)))))))


(defun MAC-mode-cleanup (list)
  "This command is called upon a MAC-list datastructure that may be
   segmented, with some MAC-strings shorter or longer than the block
   length. It attempts to make all the strings (except for the last
   one) be of MAC-length size, as well as fix up all the incorrect
   sums. Returns the value of 'h' for the list"
  (MAC-mode-fix-strings list)
  (MAC-mode-fix-sum list))

(defun MAC-mode-fix-strings (list)
  "Takes a MAC-list structure where strings may be of wrong
length, and adjusts it so all have MAC-blocksize chars, except for
possibly the last. If any block is adjusted, change the checksum
for it and the previous block to nil to indicate incorrect checksum"
  (let ((current (car (cdr list)))
(prev list)
(rest (cdr (cdr list)))
(offset 0)
(extra "")
(work-string)
(len))
    (if (not (eq (car rest) MAC-end-block))
(while (not (eq current MAC-end-block))
```

```
  (setq len (length (MAC-string current)))
  (if (= offset 0)
      (if (= len MAC-blocksize)
  (progn (setq current (car rest))
 (setq rest (cdr rest))
 (setq prev (cdr prev)))
(MAC-set-sum current 'nil)
(MAC-set-sum (car prev) 'nil)
(if (> len MAC-blocksize)
; truncate current string and
; put the rest in 'extra'
    (progn
      (setq extra (substring
   (MAC-string current) MAC-blocksize))
      (MAC-set-string
       current
       (substring (MAC-string current) 0 MAC-blocksize))
      (MAC-set-pad current (MAC-randomizer
    (MAC-string current)))
      (setq offset (- len MAC-blocksize))
      (setq current (car rest))
      (setq prev (cdr prev))
      (setq rest (cdr rest)))
; else put the entire string
; in extra, and push it into
; the next block, leaving this
; block out of the MAC-list
; structure.
  (setq extra (MAC-string current))
  (setq offset len)
  (setcdr prev rest)
  (setq current (car rest))
  (setq rest (cdr rest))))
; offset is not zero, so we
; have some extra string to
; insert into the current
; block
    (MAC-set-sum (car prev) 'nil)
    (while (> offset MAC-blocksize)  ; we need to turn as much
     ; of the 'extra' string
      ;; into new blocks as we can.
      (setcdr prev (cons (MAC-make-list-block
  (substring extra 0 MAC-blocksize))
 (cons current rest)))
      (setq extra (substring extra MAC-blocksize))
```

```
      (setq offset (- offset MAC-blocksize))
      (setq prev (cdr prev)))
    (if (= offset 0)   ; we got lucky.. exactly sized blocks
       ; were created...
'()
      (MAC-set-sum current 'nil)
      (setq work-string (concat extra (MAC-string current)))
      (if (> (length work-string) MAC-blocksize)
  (progn
    (MAC-set-string
     current (substring work-string 0 MAC-blocksize))
    (MAC-set-pad current (MAC-string current))
    (setq extra (substring work-string MAC-blocksize))
    (setq offset (length extra))
    (setq current (car rest))
    (setq rest (cdr rest))
    (setq prev (cdr prev)))
(setcdr prev rest)
(setq extra work-string)
(setq offset (length extra))
(setq current (car rest))
(setq rest (cdr rest)))))
; else we have one large block
; in between the start and end
; blocks
      (if (> (length (MAC-string current)) MAC-blocksize)
  (progn
    (setq extra (MAC-string current))
    (while (> (length extra) MAC-blocksize)
      (setcdr prev (cons
    (MAC-make-list-block
     (substring extra 0 MAC-blocksize)) rest))
      (setq extra (substring extra MAC-blocksize))
      (setq prev (cdr prev)))
    (if (> (length extra) 0)
(setcdr prev (cons (MAC-make-list-block extra) rest))
      (setcdr prev rest))))
    (while (>= offset MAC-blocksize)
      (setcdr prev (cons
    (MAC-make-list-block (substring extra 0
    MAC-blocksize))
    (cons current rest)))
      (setq prev (cdr prev))
      (setq offset (- offset MAC-blocksize))
      (setq extra (substring extra MAC-blocksize)))
```

```
    (if (> offset 0)
(setcdr prev (cons (MAC-make-list-block extra)
   (cons current rest)))))
  (setq MAC-list-modified 'nil)
  (setq MAC-list-sums-complete 'nil))

(defun MAC-mode-fix-sum (list)
  "Takes a list of MAC-list-block elements, and sets the value of
  MAC-sum wherever it is set to 'nil, otherwise, it leaves the
  current value there. returns the xor of all the sums."
  (if MAC-list-modified
      (MAC-mode-fix-strings list))
  (let ((current (car list))
(rest (cdr list))
(final 'nil))
    (while (not (eq current MAC-end-block))
      (if (null (MAC-sum current))
  (MAC-calculate-block-sum current (car rest)))
      (setq final (MAC-xor (MAC-sum current) final))
      (setq current (car rest))
      (setq rest (cdr rest)))
    final))


(defun MAC-update-buffer (buffer)
  "Attempts to fix the buffer chosen"
  (save-excursion
    (set-buffer buffer)
    (MAC-mode-fix-strings MAC-list)
    (setq MAC-list-modified 'nil)
    (setq MAC-list-sums-complete 'nil)
    (let ((current (car MAC-list))
  (rest (cdr MAC-list))
  (final 'nil))
      (while (not (eq current MAC-end-block))
(if (null (MAC-sum current))
    (MAC-calculate-block-sum current (car rest)))
(setq final (MAC-xor (MAC-sum current) final))
(setq current (car rest))
(setq rest (cdr rest)))
      (setq MAC-list-sums-complete 't)
      (setq MAC-local-h final)
      (MAC-f2 MAC-local-h)
      (setq MAC-list-final-sum-good 't))))
```

```
(defun MAC-mode-idle ()
  "The computer thinks it is being exceptionally idle, and so
is ready to compute some of the buffer"
  (if (timerp MAC-last-idle-timer)
      (cancel-timer MAC-last-idle-timer))
  (if (not (null MAC-buffers))
      (progn
(MAC-mode-incremental-fix)
(setq MAC-last-idle-timer
      (run-with-timer MAC-idle-wait
      MAC-idle-repeat
      'MAC-mode-incremental-fix)))))

(defun MAC-mode-incremental-fix ()
  "Is called often when the machine is idle, in the hopes of updating
the MAC-list datastructures of all buffers. It will do one thing at a
time, and then return, thereby being 'almost' interruptible. It also
cycles through all the buffers in MAC-buffers. There MUST be a buffer
which is in MAC-mode, or this function will fail."
  (let ((buffer))
    (setq buffer (car MAC-buffers))
    (setq MAC-buffers (cdr MAC-buffers))
    (while (and (not (null MAC-buffers))
(null (buffer-name buffer)))
      (setq buffer (car MAC-buffers))
      (setq MAC-buffers (cdr MAC-buffers)))
    (if (not (null (buffer-name buffer)))
(progn
  (if (null MAC-buffers)
      (setq MAC-buffers (list buffer))
    (nconc MAC-buffers (list buffer)))
  (save-excursion
    (set-buffer buffer)
    (if (not (null MAC-list-modified))
(progn
  (MAC-mode-fix-strings MAC-list)
  (setq MAC-list-sums-complete 'nil))
      (if (null MAC-list-sums-complete)
  (let ((current (car MAC-list))
(rest (cdr MAC-list))
(changed 'nil))
    (while (and (not (eq current MAC-end-block))
(not changed))
      (if (null (MAC-sum current))
```

```
  (progn
    (MAC-calculate-block-sum current (car rest))
    (setq changed 't)))
      (setq current (car rest))
      (setq rest (cdr rest)))
    (if (not changed)
(progn
  (setq MAC-list-sums-complete 't)
  (setq MAC-list-final-sum-good 'nil))))
(if (null MAC-list-final-sum-good)
    (let ((current (car MAC-list))
  (rest (cdr MAC-list))
  (MAC-local-h 'nil))
      (while (not (eq current MAC-end-block))
(setq MAC-local-h (MAC-xor (MAC-sum current) MAC-local-h))
(setq current (car rest))
(setq rest (cdr rest)))
      (MAC-f2 MAC-local-h)
      (setq MAC-list-final-sum-good 't)))))))))


(defun MAC-compute-buffer (b)
  "Select a buffer, and compute the MAC for it, returning the MAC as
  output"
  (save-excursion
    (set-buffer b)
    (MAC-prepare-buffer b)
    (let ((MAC (MAC-compute-from-list MAC-list)))
      (message "The MAC is %s." MAC)
      (setq MAC-list-sums-complete 't)
      (setq MAC-list-modified 'nil)
      MAC-list
      )
    )
  )

(defun MAC-prepare-buffer (b)
  "Selects a buffer, and generates a MAC list for it, without filling
in the checksums, hoping that there will be some idle-time used to do
that later"
  (save-excursion
    (set-buffer b)
    (let ((list (MAC-make-list-no-sums (buffer-string)))
      (setq MAC-list list)
      )
```

```
    )
  MAC-list
  )


; The MAC-list datastructure is a list of items
; each item contains
;     a string of MAC-blocksize letters corresponding to the block
;     a string of MAC-security-padding letters for random data
;     a variable corresponding to the current checksum of the block
(defun MAC-make-list-no-sums (str)
  "Takes a long string and returns a properly
formatted (as per MAC-list datastructure) item. It doesn't compute
the checksum for each block, instead leaving it a nil."
; because of the way lisp handles lists, it is
; easier to work backwords on the string
  (let* ((size (length str))
(blocks (/ size MAC-blocksize))
(remain (% size MAC-blocksize))
(list (cons MAC-end-block '()))
)
    (if (/= remain 0)
(setq list (cons
    (MAC-make-list-block (substring str (- 0 remain)))
    list))
      (if (= size 0) ;; empty buffer
  (setq list (cons (MAC-make-list-block "") list))))
    (while (> blocks 0)
      (setq blocks (1- blocks))
      (setq list (cons
  (MAC-make-list-block (substring str (* blocks MAC-blocksize)
  (* (1+ blocks)
     MAC-blocksize)))
  list))
      )
    (cons MAC-start-block list)
))

(defun MAC-make-list-block (str)
  "Takes a string, and turns it into a MAC-list-block.
the checksum is set to nil"
  (vector str (MAC-randomizer str)' nil))

(defun MAC-compute-h-from-list (list)
  "Takes a MAC-list datastructure, and computes the temporary h from
```

```
   it"
  (let ((current-block (car list))
(rest (cdr list))
(final))
    (MAC-mode-fix-sum list)
    (while (vectorp current-block)
      (setq final (MAC-xor final (MAC-sum current-block)))
      (setq current-block (car rest))
      (setq rest (cdr rest)))
    final))

(defun MAC-compute-from-list (list)
  "Takes a MAC-list datastructure, and computes the final MAC from it"
;;; cleanup the actual MAC-list datastructure
  (MAC-f2 (MAC-compute-h-from-list list)))

(defun MAC-calculate-block-sum (cur-block next-block)
  "Takes a current and next block, and sets the sum for the current
block using it's strings and the next block's strings,
which represents f_1(Ri, Ri+1) in the paper"
  (save-excursion
    (tq-enqueue MAC-f1-queue
(concat
 (F1-encode (concat (MAC-string cur-block)
    (MAC-pad cur-block)))
 "\n"
 (F1-encode (concat (MAC-string next-block)
    (MAC-pad next-block)))
 "\n")
".*\n" (vector cur-block (MAC-string cur-block)
       (current-buffer)) 'MAC-f1-tq-process)
    (accept-process-output MAC-f1-proc)))


(defun MAC-f1-tq-process (closure result)
  "This function is called after a transaction is completed and we
have a block checksum. This simply places the checksum in the
appropriate block"
  (save-excursion
    (set-buffer (aref closure 2))
    (if (eq (MAC-string (aref closure 0))
    (aref closure 1))     ;  check that the string hasn't
  ;  changed
(MAC-set-sum (aref closure 0) (F1-decode result))
      (MAC-set-sum (aref closure 0) 'nil))   ; if it has, reset the
```

```
     ; sum, wait to fix later,
     ; when there is more
     ; time.
    ))

(defun F1-decode (string)
  "Takes a string of the format 03 aa f2 ... and turns it
   into an array of number values"
  (let ((len (/ (length string) 3))
(result)
(current 0))
    (setq result (make-string len ?a))
    (while (< current len)
      (aset result current (str-to-num
    (substring string (* current 3)
       (+ (* current 3) 2))))
      (setq current (1+ current)))
    result))

(defun F1-encode (string)
  "Takes a string and turns it into a new string containing
   the hex representation of every char in the string, with the
   following format: '03 aa f2 '  note the final space.. it is
   necessary for this MAC-f1 program"
  (let ((len (length string))
(result "")
(current 0))
    (while (< current len)
      (setq result (concat result (format "%02x "
  (aref string current))))
      (setq current (1+ current)))
    result))

(defun str-to-num (string)
  "Takes a string of the format 'a3' or '2a' and turns it into the
actual number associated with the value"
  (let ((result 0)
(cur-string string)
(char))
    (while (> (length cur-string) 0)
      (setq char (- (string-to-char cur-string) 48))
      (if (> char 9)
  (setq char (- char 49)))
      (setq result (+ (* 16 result) char))
      (setq cur-string (substring cur-string 1)))
```

```
    result))

(defun MAC-get-f1-key ()
  "Prompts the user for the value of the key used for f1"
  (read-string
   "Please type in a key for the first-pass function (f1): ")
)

(defun MAC-get-f2-key ()
  "Prompts the user for the value of the key used for f2"
  (read-string
   "Please type in a key for the second-pass function (f2)  : ")
)

(defun MAC-xor (arg1 arg2)
  "computes the appropriate XOR of the two sum values and returns
the result"
  (if (null arg1) (copy-sequence arg2)
    (if (null arg2) (copy-sequence arg1)
      (let ((len1 (length arg1))
    (len2 (length arg2))
    (common)
    (counter 0)
    (result))
(setq result (make-string (max len1 len2) ?a))
(setq common (min len1 len2))
(while (< counter common)
  (aset result counter (logxor (aref arg1 counter)
       (aref arg2 counter)))
  (setq counter (1+ counter)))
(while (< counter len1)
  (aset result counter (aref arg1 counter))
  (setq counter (1+ counter)))
(while (< counter len2)
  (aset result counter (aref arg2 counter))
  (setq counter (1+ counter)))
result))))


(defun MAC-f2 (h)
  "Takes a value of h, and computes f2 on it"
  (save-excursion
    (tq-enqueue MAC-f2-queue (concat (F2-encode h) "\n")
".*\n" (current-buffer) 'MAC-f2-tq-process)
    (accept-process-output MAC-f2-proc))
```

```
  MAC-final-sum)

(defun F2-encode (a)
  "does encoding for string a appropriate to F2 function"
  (F1-encode a))

(defun F2-decode (a)
  "does encoding for string a appropriate to F2 function"
  (F1-decode a))

(defun MAC-f2-tq-process (closure result)
  "This function is called when the final checksum has just been
   queued for processing."
  (save-excursion
    (set-buffer closure)
    (setq MAC-final-sum (F2-decode result))))


(defun MAC-randomizer (str)
  "This functions rand(sigma), takes a string as input and computes
a resulting random pad"
  (let ((rand-string "")
(num MAC-security-padding))
    (while (> num 0)
      (setq num (1- num))
      (setq rand-string
    (concat (char-to-string (random 256))
    rand-string)))
    rand-string))

(defun MAC-valid-pad-p (str pad)
  "This function verifies whether a pad that was submitted is indeed a
   valid pad given the string input."
  't)   ; by this scheme, pads are just random data, so all are valid.


(defun MAC-save-checksum ()
  "run after a file is saved.. attempts to save the checksum"
  (if MAC-mode
      (progn
(message "attempting to save MAC.. please hold")
(MAC-update-buffer (current-buffer))
(setq MAC-orig-require-final-newline require-final-newline)
(setq require-final-newline 'nil)
(let* ((filename buffer-file-truename)
```

```
       (new-filename (concat filename ".sum"))
       (sum (prin1-to-string (MAC-compute-from-list MAC-list)))
       (str (prin1-to-string MAC-list))
       (buffer (create-file-buffer new-filename))
       )
  (save-excursion
    (switch-to-buffer buffer)
    (erase-buffer)
    (insert str ?\n sum)
    (write-file new-filename)
    (kill-buffer buffer)
    )
  (set-file-modes new-filename (file-modes filename))
  )
(setq require-final-newline MAC-orig-require-final-newline))))

(defun MAC-prepare-checksum ()
  "This function attempts to create a MAC-list datastructure that
reflects the exact status of the current buffer. This should be called
after one turns on MAC-mode to speed up the creation of the internal
datastructures by using previously saved information. Returns a
MAC-list datastructure that reflects the current buffer's state"
  (let ((list)
(pointer)
(m)
(sumfile)
(buffer)
)
      (if (null buffer-file-name)
  (progn
    (message
     "Could not find checksum. generating. please wait")
    (setq list (MAC-prepare-buffer (current-buffer))))
(setq sumfile (concat buffer-file-name ".sum"))
(if (not (file-readable-p sumfile))
    (progn
      (message
       "Could not read checksum. generating. please wait")
      (setq list (MAC-prepare-buffer (current-buffer))))
  (setq buffer (find-file-noselect sumfile))
  (setq m (set-marker (make-marker) 1 buffer))
  (setq list (read m))
  (setq pointer list)
  (setq MAC-start-block (car list))
  (while (and (not (null pointer))
```

```
      (not (equal (car (cdr pointer))
  MAC-end-block)))
    (setq pointer (cdr pointer)))
  (if (null pointer)
      (progn
(message
 "sumfile is incorrect, generating, please wait")
(kill-buffer buffer)
(setq list (MAC-prepare-buffer (current-buffer))))
    (setcdr pointer (cons MAC-end-block '()))
    (if (MAC-list-matches-buffer-p list (current-buffer))
(message "Found sumfile with good checksum. using")
      (message
       "sumfile is incorrect, generating, please wait")
      (setq list (MAC-prepare-buffer (current-buffer)))))
  (kill-buffer buffer)))
      (setq MAC-list-modified 't)
      list))

(defun MAC-list-matches-buffer-p (list buffer)
  "Takes a MAC-list structure and a buffer, and checks whether the two
agree on state. THIS DOES NOT CHECK THE CHECKSUM VALUES, only that the
strings in the buffer and in the MAC-list are the same."
  (let ((current (car list))
(rest (cdr list))
(temp-buffer)
(max-for-real)
(max-for-temp)
(valid 't)
)
    (setq temp-buffer (generate-new-buffer "temp"))
    (save-excursion
      (if (not (eq current MAC-start-block))
  (setq valid 'nil))
      (setq current (car rest))
      (setq rest (cdr rest))
      (set-buffer buffer)
      (setq max-for-real (point-max))
      (set-buffer temp-buffer)
      (while (and (and valid (not (eq current 'nil)))
  (not (eq current MAC-end-block)))
(insert (MAC-string current))
(setq current (car rest))
(setq rest (cdr rest))
)
```

```
      (setq max-for-temp (point-max))
      (if (not
   (eq 0 (compare-buffer-substrings
  buffer 1 max-for-real temp-buffer 1 max-for-temp)))
  (setq valid 'nil))
      (if (not (eq current MAC-end-block))
  (setq valid 'nil))
      )
    (kill-buffer temp-buffer)
    valid))

(defun valid-block-p (block next)
  "Evaluates whether a block is valid(i.e. the checksum and
              the pad match the blocks' data)"
  (if (null block)
      'nil
    (if (and (null next) (not (eq next MAC-end-block)))
'nil
      (let ((valid 't)
    (first (copy-sequence block)))
(MAC-calculate-block-sum first next)
(if (not (MAC-valid-pad-p (MAC-string block) (MAC-pad block)))
    (setq valid 'nil)
  (if (not (equal (MAC-sum first) (MAC-sum block)))
      (setq valid 'nil)))
valid))))


(defun MAC-check-file (filename)
  "Checks the MAC of a supplied filename for validity."
  (interactive "fFilename to check the MAC of : ")
  (save-excursion
    (let ((list)
  (final-sum)
  (computed-final-sum)
  (pointer)
  (m)
  (sumfile)
  (databuffer)
  (datapoint 1)
  (buffer)
  (valid 'nil)
  (done 'nil)
  )
      (if (null filename)
```

```
  (message "Could not find %s -- BAD checksum" filename)
(setq sumfile (concat filename ".sum"))
(if (not (file-readable-p sumfile))
    (message
     "Could not read checksum: %s.sum --  BAD" filename)
  (setq valid 't)
  (setq databuffer (find-file-read-only filename))
  (if (null MAC-f1-key)
      (progn
(setq MAC-f1-key (MAC-get-f1-key))
(let ((process-connection-type nil))
  (setq MAC-f1-proc (start-process "F1"
   (generate-new-buffer-name " F1")
   "/i/yoav/bin/MAC-f1")))
(process-send-string MAC-f1-proc (concat MAC-f1-key "\n"))
(setq MAC-f1-queue (tq-create MAC-f1-proc))))
  (if (null MAC-f2-key)
      (progn
(setq MAC-f2-key (MAC-get-f2-key))
(let ((process-connection-type nil))
  (setq MAC-f2-proc (start-process "F2"
   (generate-new-buffer-name " F2")
   "/i/yoav/bin/MAC-f2")))
(process-send-string MAC-f2-proc (concat MAC-f2-key "\n"))
(setq MAC-f2-queue (tq-create MAC-f2-proc))))
  (setq buffer (find-file-noselect sumfile))
  (setq m (set-marker (make-marker) 1 buffer))
  (setq list (read m))
  (setq final-sum (read m))
  (setq pointer (car list))
  (setq list (cdr list))
  (if (null (cdr list))
      (setq valid 'nil)
    (if (not (string= (MAC-string pointer)
      (MAC-string MAC-start-block)))
(setq valid 'nil))
    (if (not (string= (MAC-pad pointer)
      (MAC-pad MAC-start-block)))
(setq valid 'nil))
    (if (not (valid-block-p pointer (car list)))
(setq valid 'nil))
    (setq computed-final-sum (MAC-sum pointer)))
  (while (and valid (not done))
    (setq pointer (car list))
    (setq computed-final-sum
```

```
  (MAC-xor computed-final-sum (MAC-sum pointer)))
    (setq list (cdr list))
    (if (null list)
(setq done 't)
      (if (not
   (string=
    (buffer-substring datapoint
      (+ datapoint
 (length (MAC-string pointer))))
    (MAC-string pointer)))
  (setq valid 'nil)
(setq datapoint (+ datapoint (length (MAC-string pointer))))
(if (not (valid-block-p pointer (car list)))
    (setq valid 'nil)))))
  (if done
      (if (not (and (string= (MAC-string pointer)
     (MAC-string MAC-end-block))
    (string= (MAC-pad pointer)
     (MAC-pad MAC-end-block))))
  (setq valid 'nil)
(if (not (equal (MAC-f2 computed-final-sum) final-sum))
    (setq valid 'nil))))
  (tq-close MAC-f1-queue)
  (tq-close MAC-f2-queue)
  (kill-buffer buffer)
  (kill-buffer databuffer)
  (if valid (message "Good Checksum for %s." filename)
    (message "BAD Checksum for %s." filename))
  valid)))))

(defun MAC-percent (&optional arg)
  "Computes the percentage of the buffer for which a valid MAC has
been computed. This is useful for timing and optimization tests."
  (interactive "b")
  (save-excursion
    (set-buffer arg)
    (let ((good 0)
  (bad 0)
  (current (car MAC-list))
  (rest (cdr MAC-list)))
      (while (not (equal MAC-end-block current))
(if (null (MAC-sum current))
    (setq bad (+ bad 1))
  (setq good (+ 1 good)))
(setq current (car rest))
```

```
(setq rest (cdr rest)))
      (message "The percentage is %s."
       (/ (* 100 good) (+ good bad))))))
```

A.1.2 MAC-f1

This is C code to do faster computations. It currently use DES in ECB mode.

```
/*  MAC-f1
 *    This file contains an implementation of whatever it is that
 *    MAC-f1 is deemed to do. In this case, MAC-f1 simply takes an
 *    arbitrary length string and calculates DES(string), returning
 *    that as output.  for security reasons, the program can't be
 *    called with the key as argument, so instead, it is passed the
 *    key in as the first string, and all the following strings are
 *    encrypted under that key.
 *
 *  Written by Yoav Yerushalmi as part of the MAC.el package, see
 *  comments in MAC.el for further info.
 */

#include <des.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

#define MAX_BLOCK_LEN 256
        /* maximum size of input in any one string */

#define   min(a,b)        ((a) < (b) ? (a) : (b))

void outfun(unsigned char);

int main() {
  des_cblock key;
  des_cblock input, output;
  des_key_schedule schedule;
  char str1[MAX_BLOCK_LEN + 1];
  char str2[MAX_BLOCK_LEN + 1];
  unsigned char temp[MAX_BLOCK_LEN + 1];
  unsigned char out1[MAX_BLOCK_LEN + 1];
  unsigned char out2[MAX_BLOCK_LEN + 1];
```

```
  int ref, outputlen;
  int len1, len2, shorter;


  signal(SIGHUP, exit);
  if (!fgets(str1, MAX_BLOCK_LEN, stdin)) {
    perror("failed to get key : ");
    exit(1);
  }
  des_string_to_key(str1, &key);
  if (key_sched(&key, schedule)) {
    fprintf(stderr, "MAC-f1: Failed to set keyschdule.. exiting\n");
    exit(1);}
  while (1) {
    if ((fgets(str1, MAX_BLOCK_LEN, stdin)) &&
(fgets(str2, MAX_BLOCK_LEN, stdin))) {
      len1 = strlen(str1) -1;
      len2 = strlen(str2) -1;
      len1 = (len1 / 3);
      len2 = (len2 / 3);
      ref = 0;
      while (ref < len1) {
temp[ref] = (unsigned char)strtol(&str1[(ref * 3)], NULL, 16);
ref++;
      }
      memcpy(str1, temp, ref);
      str1[len1] = '\0'; /* remove the trailing newline */

      ref = 0;
      while (ref < len2) {
temp[ref] = (unsigned char)strtol(&str2[(ref * 3)], NULL, 16);
ref++;
      }
      memcpy(str2, temp, ref);
      str2[len2] = '\0'; /* remove the trailing newline */

      /*  Calculate DES(str1)  */
      len1 -= 8;
      ref = 0;
      while (ref <= len1) {
memcpy(input, &str1[ref], 8);
des_ecb_encrypt(&input, &output, schedule, 1);
memcpy(&out1[ref], output, 8);
ref += 8;
      }
```

```
      len1 += 8;
      if (ref < len1) {
des_string_to_key(&str1[ref], &input);
des_ecb_encrypt(&input, &output, schedule, 1);
memcpy(&out1[ref], output, 8);
ref += 8;
      }
      out1[ref] = '\0';

      /* Calculate DES(str2)  */
      len2 -= 8;
      ref = 0;
      while (ref <= len2) {
memcpy(input, &str2[ref], 8);
des_ecb_encrypt(&input, &output, schedule, 1);
memcpy(&out2[ref], output, 8);
ref += 8;
      }
      len2 += 8;
      if (ref < len2) {
des_string_to_key(&str2[ref], &input);
des_ecb_encrypt(&input, &output, schedule, 1);
memcpy(&out2[ref], output, 8);
ref += 8;
      }
      out2[ref] = '\0';

      /* Calculate DES(str1) (xor) DES(str2)  */
      shorter = min(len1, len2);
      outputlen = shorter + ((8 - (shorter % 8)) % 8);
      for (ref=0; ref<outputlen; ref++)
outfun(out1[ref] ^ out2[ref]);
      if (len1>len2)
while (ref < (len1 + ((8 - (len1 % 8)) % 8))) {
  outfun(out1[ref]);
  ref++;
}
      if (len2 > len1)
while (ref < (len2 + ((8 - (len2 % 8)) % 8))) {
  outfun(out2[ref]);
  ref++;
}
      printf ("\n");
      fflush(stdout);
    }
```

```
    else
      fprintf(stderr, "Problems");
  }
}

void outfun(unsigned char foo)
  {
    printf("%02x ", foo);
  }
```

A.1.3 MAC-f2

This is the same as f1(), but only has one real input. Again it uses DES in ECB mode.

```
/*  MAC-f2
 *    This file contains an implementation of whatever it is that
 *    MAC-f1 is deemed to do. In this case, MAC-f1 simply takes an
 *    arbitrary length string and calculates DES(string), returning
 *    that as output. for security reasons, the program can't be
 *    called with the key as argument, so instead, it is passed the
 *    key in as the first string, and all the following strings are
 *    encrypted under that key.
 *
 *  Written by Yoav Yerushalmi as part of the MAC.el package, see
 *  comments in MAC.el for further info.
 */

#include <des.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

#define MAX_BLOCK_LEN 256
        /* maximum size of input in any one string */

#define   parse(a)        ((16 * (a)[0]) + (a)[1])

void outfun(unsigned char);

int main() {
```

```
des_cblock key;
des_cblock input, output;
des_key_schedule schedule;
char str[MAX_BLOCK_LEN+1];
unsigned char temp[MAX_BLOCK_LEN + 1];
unsigned char out[MAX_BLOCK_LEN+1];
int ref;
int len, outputlen;

signal(SIGHUP, exit);  /* terminate when we get sighup */
if (!fgets(str, MAX_BLOCK_LEN, stdin)) {
  perror("MAC-f2 failed to get key : ");
  exit(1);
}
des_string_to_key(str, &key);
if (key_sched(&key, schedule)) {
  fprintf(stderr, "MAC-f2: Failed to set keyschdule.. exiting\n");
  exit(1);}
while (fgets(str, MAX_BLOCK_LEN, stdin)) {
  len = strlen(str) - 1;
  len = (len / 3);
  ref = 0;
  while (ref < len) {
    temp[ref] = (unsigned char)strtol(&str[(ref * 3)], NULL, 16);
    ref++;
  }
  memcpy(str, temp, ref);
  str[len] = '\0'; /* remove the trailing newline */
  outputlen = 0;
  len -= 8;
  ref = 0;
  while (ref <= len) {
    outputlen+=8;
    memcpy(input, &str[ref], 8);
    des_ecb_encrypt(&input, &output, schedule, 1);
    memcpy(&out[ref], output, 8);
    ref += 8;
  }
  if (ref < (len+8)) {
    outputlen += 8;
    des_string_to_key(&str[ref], &input);
    des_ecb_encrypt(&input, &output, schedule, 1);
    memcpy(&out[ref], output, 8);
  }
  for (ref=0; ref < outputlen; ref++)
```

```
      outfun(out[ref]);
    printf("\n");
    fflush(stdout);
  }
}


void outfun(unsigned char foo)
{
  printf("%02x ", foo);
}
```

A.2 usage.el

The following code is used to examine the usage patterns of the person typing at the keyboard. Using these numbers, it becomes easier to pick the appropriate algorithm or variables to achieve best performance.

```
;;; Code to figure out what kind of changes a person makes when
;;;   editing a buffer. This will attempt to generate variables which
;;;   contain the number of all sorts of changes (see the variable
;;;   descriptions).

;;; Written by Yoav Yerushalmi as part of his thesis research.
;;;   Supervised by Professor Shafi Goldwasser
;;;
;;;    March 20, 1997
;;;
;;; Permission is granted to use and modify the code as desired as
;;;  long as credit is given to the above people.


(require 'timer)        ; need the timer package for idleness

(defvar total-changes 0
  "The total number of changes made to the document")
(defvar insert 0
  "The total number of inserts made to the document (of any length)")
(defvar delete 0
  "the total number of deletes")
(defvar modify 0
  "the total number of modify operations")
(defvar idle5 0
  "the total number of times the buffer was idle for over half a
   second")
(defvar idle10 0
  "the total number of times the buffer was idle for over a second")
(defvar idle20 0
  "the total number of times the buffer was idle for over two
   seconds")
(defvar idle100 0
  "the total number of times the buffer was idle for over ten
   seconds")
(defvar start-time 0
```

```
  "the time from when the use-mode was turned on")

;; These variables will be used to figure out what kind of above
;; operations were made
(defvar append 0
  "an insert at the end")
(defvar truncate 0
  "a deletion at the end")
(defvar ins-1qr 0
  "an insertion in the 1st quarter of the document")
(defvar ins-2qr 0
  "an insertion in the 2nd quarter of the document")
(defvar ins-3qr 0
  "an insertion in the 3rd quarter of the document")
(defvar ins-4qr 0
  "an insertion in the 4th quarter of the document")
(defvar del-1qr 0
  "a deletion in the 1st quarter of the document")
(defvar del-2qr 0
  "a deletion in the 2nd quarter of the document")
(defvar del-3qr 0
  "a deletion in the 3rd quarter of the document")
(defvar del-4qr 0
  "a deletion in the 4th quarter of the document")

(make-variable-buffer-local 'total-changes)
(make-variable-buffer-local 'insert)
(make-variable-buffer-local 'delete)
(make-variable-buffer-local 'modify)
(make-variable-buffer-local 'idle5)
(make-variable-buffer-local 'idle10)
(make-variable-buffer-local 'idle20)
(make-variable-buffer-local 'idle100)
(make-variable-buffer-local 'start-time)

(make-variable-buffer-local 'append)
(make-variable-buffer-local 'truncate)
(make-variable-buffer-local 'ins-1qr)
(make-variable-buffer-local 'ins-2qr)
(make-variable-buffer-local 'ins-3qr)
(make-variable-buffer-local 'ins-4qr)
(make-variable-buffer-local 'del-1qr)
(make-variable-buffer-local 'del-2qr)
(make-variable-buffer-local 'del-3qr)
(make-variable-buffer-local 'del-4qr)
```

```
(defvar use-mode-orig-after-change-functions 'nil
  "original value of after-change-functions")

(defvar idle-timers-to-cancel 'nil)

(defvar use-mode 'nil
  "*Non-nil enables usage mode on buffer")
(make-variable-buffer-local 'use-mode)

(or (assq 'use-mode minor-mode-alist)
    (setq minor-mode-alist (cons '(use-mode " use")
 minor-mode-alist)))


(defun use-mode (&optional arg)
  "minor mode for figuring out the kinds of usage on a buffer"
  (interactive "P")
  (setq use-mode
(if (null arg)
    (not use-mode)
  (> (prefix-numeric-value arg) 0)))
  (if use-mode
      (progn
(setq use-mode-orig-after-change-functions
      after-change-functions)
;reset variables
(setq start-time (current-time))
(setq total-changes 0)
(setq insert 0)
(setq delete 0)
(setq modify 0)
(setq idle5 0)
(setq idle10 0)
(setq idle20 0)
(setq idle100 0)
(setq append 0)
(setq truncate 0)
(setq ins-1qr 0)
(setq ins-2qr 0)
(setq ins-3qr 0)
(setq ins-4qr 0)
(setq del-1qr 0)
(setq del-2qr 0)
(setq del-3qr 0)
```

```
(setq del-4qr 0)

(setq idle-timers-to-cancel
      (cons (run-with-idle-timer 0.5 't 'use-mode-5-dsec)
    idle-timers-to-cancel))
(setq idle-timers-to-cancel
      (cons (run-with-idle-timer 1   't 'use-mode-10-dsec)
    idle-timers-to-cancel))
(setq idle-timers-to-cancel
      (cons (run-with-idle-timer 2   't 'use-mode-20-dsec)
    idle-timers-to-cancel))
(setq idle-timers-to-cancel
      (cons (run-with-idle-timer 10  't 'use-mode-100-dsec)
    idle-timers-to-cancel))
(setq after-change-functions (cons
      'use-mode-buffer-change-hook
      after-change-functions)))
    (setq after-change-functions
  use-mode-orig-after-change-functions)))

(defvar use-mode-map 'nil "Keymap for use-mode.")

(if (null use-mode-map)
    (fset 'use-mode-map
  (setq use-mode-map
(copy-keymap (current-global-map)))))

(if (not (assq 'use-mode minor-mode-map-alist))
    (setq minor-mode-map-alist
  (cons (cons 'use-mode use-mode-map)
minor-mode-map-alist)))

(defun use-mode-5-dsec ()
  "Is called after the user hasn't done anything for half a second"
  (setq idle5 (1+ idle5))
  )
(defun use-mode-10-dsec ()
  "Is called after the user hasn't done anything for a second"
  (setq idle10 (1+ idle10))
  (setq idle5 (1- idle5))
  )
(defun use-mode-20-dsec ()
  "Is called after the user hasn't done anything for two seconds"
  (setq idle10 (1- idle10))
  (setq idle20 (1+ idle20))
```

```
)
(defun use-mode-100-dsec ()
  "Is called after the user hasn't done anything for ten seconds"
  (setq idle20 (1- idle20))
  (setq idle100 (1+ idle100))
)


(defun use-mode-buffer-change-hook (beg end old-len)
  "This function is used to keep track of the kinds of changes being
  made to the document. It is called every time a change to the buffer
  is made, whether an insert, delete, or modify. It then updates the
  appropriate variables"
  (let* ((new-len (- end beg))
(buffer-len (length (buffer-string)))
(quarter (/ buffer-len 4))
)
    (setq total-changes (1+ total-changes))
    ;; modification
    (if (= new-len old-len)
(setq modify (1+ modify))
      ;; deletion
      (if (> old-len new-len)
  (progn
    (setq delete (1+ delete))
    (if (> end buffer-len)
(progn
  (setq truncate (1+ truncate))
  (setq del-4qr (1+ del-4qr)))
      (if (< beg quarter)
  (setq del-1qr (1+ del-1qr))
(if (< beg (* 2 quarter))
    (setq del-2qr (1+ del-2qr))
  (if (< beg (* 3 quarter))
      (setq del-3qr (1+ del-3qr))
    (setq del-4qr (1+ del-4qr))))))
;; insertion
(setq insert (1+ insert))
(if (= beg buffer-len)
    (progn
      (setq append (1+ append))
      (setq ins-4qr (1+ ins-4qr)))
  (if (< beg quarter)
      (setq ins-1qr (1+ ins-1qr))
    (if (< beg (* 2 quarter))
```

```
(setq ins-2qr (1+ ins-2qr))
      (if (< beg (* 3 quarter))
  (setq ins-3qr (1+ ins-3qr))
(setq ins-4qr (1+ ins-4qr))))))))))




(defun use-report (&optional arg)
  (interactive "b")
  (let ((time (current-time))
(total-time))

    (setq total-time (* 65536 (- (car time) (car start-time))))
    (setq total-time (+ total-time
(- (car (cdr time))
   (car (cdr start-time)))))
    (message "-----------------------------------------------------")
    (message "   Use-mode report for buffer %s" (buffer-name))
    (message "-----------------------------------------------------")
    (message "Total number of changes made :   %s" total-changes)
    (message "                     inserts :   %s %%"
     (/ (* 100 insert) total-changes))
    (message "                     deletes :   %s %%"
     (/ (* 100 delete) total-changes))
    (message "               modifications :   %s %%"
     (/ (* 100 modify) total-changes))
    (message "                     appends :   %s %%"
     (/ (* 100 append) insert))
    (message "                   truncates :   %s %%"
     (/ (* 100 truncate) delete))
    (message " the insertions occurred in:")
    (message "                 1st quarter :   %s %%"
     (/ (* 100 ins-1qr) insert))
    (message "                 2nd quarter :   %s %%"
     (/ (* 100 ins-2qr) insert))
    (message "                 3rd quarter :   %s %%"
     (/ (* 100 ins-3qr) insert))
    (message "                 4th quarter :   %s %%"
     (/ (* 100 ins-4qr) insert))
    (message " the deletions occurred in:")
    (message "                 1st quarter :   %s %%"
     (/ (* 100 del-1qr) delete))
    (message "                 2nd quarter :   %s %%"
     (/ (* 100 del-2qr) delete))
```

```
 (message "                 3rd quarter :   %s %%"
  (/ (* 100 del-3qr) delete))
 (message "                 4th quarter :   %s %%"
  (/ (* 100 del-4qr) delete))
 (message "")
 (message " The editing of the buffer took %s seconds."
  total-time)
 (message " During the editing of the buffer, it was idle for")
 (message "     between 1/2 sec to a sec :  %s times" idle5)
 (message "      between 1 sec to 2 secs :  %s times" idle10)
 (message "  between 2 secs to a 10 secs :  %s times" idle20)
 (message "                 over 10 secs :  %s times" idle100)
 (message "-----------------------------------------------------")
))
```

A.3 Incremental Public Key Encryption Code

The following contains all the source code used to generate an incremental public-key encryption of a buffer in emacs. For mmore details on this code, refer to chapter 5

A.3.1 PKE.el

This elisp code is used to create the minor-mode in emacs, in a similar way to the minor mode for MACs listed earlier.

```
;;;  Code to create a minor mode that will notice when the buffer is
;;;  edited, and appropriately modify an encryption of the doc using
;;;  a given key.
;;;
;;;  The algorithms are described in the thesis "Incremental
;;;  Cryptography by Yoav Yerushalmi, as part of the requirements for
;;;  the degree of Masters of Engineering in Computer Science at
;;;  M.I.T.
;;;
;;;  Written by Yoav Yerushalmi under the supervision of Prof. Shafi
;;;  Goldwasser.  May 20, 1997
;;;
;;;  Permission is granted to use and modify this code as desired, as
;;;  long as credit is given to all the above.

;;; -----------------------------------------------------------------
;;; PKE Mode An elisp minor mode to generate an incremental encryption
;;; of a buffer using a public key specified.
;;;
;;;   To use, simply run PKE-mode on a buffer, and edit it. When
;;;   trying to save, the encryption will be written out
;;;   automatically to a file named {foo}.pke
;;;
;;;   This is still NOT a complete product. Please understand many
;;;   things still will change.
;;;
;;;  Finally, a note. This scheme works by dealing only with
;;;  blocksizes of size 1 (unlike the MAc mode which had variable
;;;  blocksizes). While it is possible and feasible to analyze the
;;;  behavior of larger blocks, a blocksize of one, which generates
;;;  the largest encryption, is also the cheapest to increment.
```

```
;;;
;;;  At some point in the future, therefore, this may add code to deal
;;;  with larger blocksizes.

(defconst PKE_PROC "/mit/yoav/work/lcs/bin/PKE"
  "the actual process that does the encrypting and processing")

(random t)

(defvar PKE-key 'nil
  "The key being used for the current buffer encryption")
(defvar PKE-proc 'nil
  "The process associated with the current buffer's encryption")
(defvar PKE-output ""
  "the string associated with the output from the process")
(defvar PKE-mode 'nil
  "*Non-nil enables Public-key encryption mode for the buffer. Default
is nil. To change the value do:
(set-default 'PKE-mode 't)")

(make-variable-buffer-local 'PKE-key)
(make-variable-buffer-local 'PKE-proc)
(make-variable-buffer-local 'PKE-output)
(make-variable-buffer-local 'PKE-mode)

(make-variable-buffer-local 'after-save-hooks)
(make-variable-buffer-local 'after-change-functions)

(or (assq 'PKE-mode minor-mode-alist)
    (setq minor-mode-alist
  (cons '(PKE-mode " PKE") minor-mode-alist)))

(defun PKE-find-file-hook ()
  "hook to call when a file is being loaded to tell if it has been
saved with a PKE"
   (let* ((name buffer-file-truename)
  (sumname (concat name ".pke")))
     (if (file-readable-p sumname) (PKE-mode))))

(defun PKE-kill-buffer-hook ()
  "hook to call when a buffer is killed"
  (if (not (null PKE-proc))
      (progn
(PKE-send "quit\n")
(delete-process PKE-proc))))
```

```
(add-hook 'find-file-hooks 'PKE-find-file-hook)
(add-hook 'kill-buffer-hooks 'PKE-kill-buffer-hook)

(defun PKE-mode (&optional arg)
  "Minor mode for editing buffers and generating public key
encryptions of the buffer in the background."
  (interactive "P")
  (setq PKE-mode
(if (null arg)
    (not PKE-mode)
  (> (prefix-numeric-value arg) 0)))
  (if PKE-mode
      (progn
(if (null PKE-key)
    (progn
      (setq PKE-key (PKE-get-key))
      (let ((process-connection-type 'nil))
(setq PKE-proc
      (start-process "PKE"
     (generate-new-buffer-name " PKE")
     PKE_PROC))
(if (null PKE-proc)
    (error "Problems starting up " PKE_PROC))
(process-kill-without-query PKE-proc))
      (set-process-filter PKE-proc 'PKE-filter)))
(add-hook 'after-save-hook 'PKE-save)
(setq after-change-functions
      (cons 'PKE-mode-buffer-change-hook after-change-functions))
(PKE-initiate))
    (delete 'PKE-mode-buffer-change-hook after-change-functions)
    (remove-hook 'after-save-hook 'PKE-save)
    (setq PKE-key 'nil)
    (PKE-terminate)))

(defvar PKE-mode-map 'nil "Keymap for PKE mode")

(if (null PKE-mode-map)
    (fset 'PKE-mode-map
  (setq PKE-mode-map (copy-keymap (current-global-map)))))

(if (not (assq 'PKE-mode minor-mode-map-alist))
    (setq minor-mode-map-alist
  (cons (cons 'PKE-mode PKE-mode-map)
minor-mode-map-alist)))
```

```
(defun PKE-mode-buffer-change-hook (beg end old-len)
  "This function notices changes to the buffer, and informs the PKE.c
process about them"
  (let ((temp)
(message))
    (if (and (= old-len 0)
     (< beg end))
; We have an insertion
(progn
  (setq temp beg)
  (while (< temp end)
    (PKE-send "Insert:\n")
    (PKE-send (format "%d\n" temp))
    (PKE-send (concat (PKE-encode
       (buffer-substring temp (1+ temp)))
      "\n"))
    (setq temp (1+ temp))))
      (if (and (> old-len 0)
       (= beg end))
; We have a deletion.
  (progn
    (setq temp old-len)
    (while (> temp 0)
      (PKE-send "Delete:\n")
      (PKE-send (format "%d\n" beg))
      (setq temp (1- temp))))
; We have a modification.
(setq temp beg)
(while (< temp end)
  (PKE-send "Modify:\n")
  (PKE-send (format "%s\n" temp))
  (PKE-send (concat (PKE-encode
     (buffer-substring temp (1+ temp)))
    "\n"))
  (setq temp (1+ temp)))))))

(defun PKE-initiate ()
  "Initiate communications with the associated PKE.c process"

  (let* ((name buffer-file-truename)
 (sumname (concat name ".pke")))
    (PKE-send (concat "Public-key:\n" PKE-key "\n"))
    (if (and
```

```
 (file-readable-p sumname)
 (newer-file sumname name))
(PKE-send (concat "Load:\n" sumname "\n"))
      (PKE-send "Create:\n")
      (PKE-send-buffer)
      (PKE-send "End\n"))))

(defun PKE-terminate ()
  "Things to do to cleanup after we quit PKE mode"

  ;;  Cleanup whatever is left...?
  ;;
  ;; todo
  ;;
)


(defun PKE-send-buffer ()
  "Sends the current buffer encoded as per pkeapi spec"
  (let* ((string (buffer-string))
(len (length string))
(ptr 0))
    (while (< ptr len)
      (PKE-send (concat (number-to-string (aref string ptr)) " \n"))
      (setq ptr (1+ ptr)))))

(defun PKE-decode (string)
  "Takes a string of the format 03 aa f2 ... and turns it
   into an array of number values"
  (let ((len (/ (length string) 3))
(result)
(current 0))
    (setq result (make-string len ?a))
    (while (< current len)
      (aset result current (str-to-num
    (substring string (* current 3)
       (+ (* current 3) 2))))
      (setq current (1+ current)))
    result))

(defun str-to-num (string)
  "Takes a string of the format 'a3' or '2a' and turns it into the
actual number associated with the value"
  (let ((result 0)
(cur-string string)
```

```
(char))
    (while (> (length cur-string) 0)
      (setq char (- (string-to-char cur-string) 48))
      (if (> char 9)
  (setq char (- char 49)))
      (setq result (+ (* 16 result) char))
      (setq cur-string (substring cur-string 1)))
    result))

(defun PKE-encode (string)
  "Takes a string and turns it into a new string containing
   the hex representation of every char in the string, with the
   following format: '03 aa f2 '  note the final space."
  (let ((len (length string))
(result "")
(current 0))
    (while (< current len)
      (setq result (concat result (format "%02x "
  (aref string current))))
      (setq current (1+ current)))
    result))


(defun PKE-get-key ()
  "Prompts user for public key to use"
  (read-string
   "Please type in the name of the public-key to use : "))

(defun PKE-send (string)
  "sends the exact content of 'string' to the associated process."
  (process-send-string PKE-proc string))

(defun PKE-receive (&optional regexp)
  "waits for receiving data from process. We have complete data when
we find a '\n'"
  (save-match-data
    (let ((begin (string-match regexp PKE-output))
  (result)
  (end))
      (while (null begin)
(accept-process-output PKE-proc)
(setq begin (string-match regexp PKE-output)))
      (setq end (match-end 0))
      (setq result (substring PKE-output begin end))
      (setq PKE-output (substring PKE-output end))
```

```
      result)))

(defun PKE-save ()
  "responsible for getting the document's other component saved"

  (if PKE-mode
      (let* ((name buffer-file-truename)
     (sumname (concat name ".pke\n")))
(PKE-send "Save:\n")
(PKE-send sumname)
(PKE-receive ".*")
(message "Done saving checksum")
)))

(defun PKE-filter (process output)
  "The filter that is responsible for building the string that will be
returned as the result for PKE-receive"
  (setq PKE-output (concat PKE-output output)))


(defun newer-file (file1 file2)
  "Checks the mod-time on the two files and decides which is the newer
file"
  (if (not (file-readable-p file1))
      'nil
    (if (not (file-readable-p file2))
't
      (let ((file1time (car (nthcdr 5 (file-attributes file1))))
    (file2time (car (nthcdr 5 (file-attributes file2)))))
(if (> (car file1time) (car file2time))
    't
  (if (= (car file1time) (car file2time))
      (if (< (car (cdr file1time)) (car (cdr file2time)))
  'nil
't)
    'nil))))))
```

A.3.2 PKE.c

This c code implements all the actual computations and the interface exported to **PKE.el**. It is implemented in C for efficiency, and works as a module, allowing other programs to replace it when necessary. This code uses the RSAREF package (see

[15]) for more documentation.

PKE.c

The main source-code file:

```
/*  PKE.c
 *    The code which implements all the routines necessary to
 * support the PKE.el routines (see pkeapi). It makes extensive use of
 * the RSAREF implementation from RSA Laboratories.
 *
 *  The datastructure used to represent this is a linked list of
 *  pke_elt's, where each pke_elt holds a pair of values:
 *      data   :   the encrypted item (random + plaintext).
 *      next   :   a pointer to the next element in the list
 *                    or NULL for the last element.
 */


#include "PKE.h"

#include <stdio.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <pwd.h>

/* Some static variables.... these should be constant throughout
  the program */
R_RSA_PUBLIC_KEY key;
int key_good = 0;
R_RANDOM_STRUCT randomStruct;
pke_elt *head = NULL; /* The head of the list */
int current_num = 0;         /* the number of the element we just
looked at */
pke_elt *current_ptr; /* a cache pointer to the element we just
dealt with */
char input[MAX_INPUT_SIZE];  /* Note static size!! bad!! */
```

```
int main_loop(), get_input();
void set_key(), reset_structure(), load_file(), create(),
  modify(), delete(), insert(), save_file(),
  decode(char *, char *), PKE_encrypt(char *, int, pke_elt *),
  pad_with_random(char *), get_home(char *), Init_setup();


int main(argc, argv){
  /* Setup whatever is necessary... */

  if (argc > 1) {
    fprintf(stderr,
    "This program has NO arguments. See the documentation!\n");
    exit(1);
  }
  Init_setup();

  return(main_loop());
}

int main_loop() {
  /* loop in this function until the input is 'End' and arbitrate what
     to do based on input received here */

  int finished = 0;

  while (!finished){
    switch(get_input()) {
    case INPUT_QUIT:
      finished = 1;
      break;
    case INPUT_KEY:
      reset_structure();
      set_key();
      break;
    case INPUT_CREATE:
      create();
      break;
    case INPUT_INSERT:
      insert();
      break;
    case INPUT_MODIFY:
      modify();
```

```
      break;
    case INPUT_DELETE:
      delete();
      break;
    case INPUT_SAVE:
      save_file();
      break;
    case INPUT_LOAD:
      load_file();
      break;
    case INPUT_ERROR:
      fprintf(stderr, "Got something bad: %s\n", input);
      break;
    default:
      fprintf(stderr, "I'm confused : %s\n", input);
      exit(1);
    }
  }
  return (0);
}

int get_input() {
  /* Figure out what needs to be done */

  if ((fgets(input, MAX_INPUT_SIZE, stdin)) == NULL)
    return (INPUT_ERROR);
  if (! strncasecmp(input, "public-key",
    (sizeof("public-key") - 1)))
    return(INPUT_KEY);
  if (! strncasecmp(input, "create", (sizeof("create") - 1)))
    return(INPUT_CREATE);
  if (! strncasecmp(input, "insert", (sizeof("insert") - 1)))
    return (INPUT_INSERT);
  if (! strncasecmp(input, "modify", (sizeof("modify") - 1)))
    return (INPUT_MODIFY);
  if (! strncasecmp(input, "delete", (sizeof("delete") - 1)))
    return (INPUT_DELETE);
  if (! strncasecmp(input, "save", (sizeof("save") - 1)))
    return (INPUT_SAVE);
  if (! strncasecmp(input, "load", (sizeof("load") - 1)))
    return (INPUT_LOAD);
  if (! strncasecmp(input, "quit", (sizeof("quit") - 1)))
    return (INPUT_QUIT);
  return(INPUT_ERROR);
}
```

```
void decode(char *input, char *output){
  /* decode a string of form "42 6a" (hex ascii codes) into
     appropriate format..  assumes input is a string with enough space
     to hold result */
  int len, ref_i, ref_o;

  len = strlen(input)/3;
  ref_i = 0;
  ref_o = 0;
  while (ref_i < len){
    output[ref_o] = (unsigned char)strtol(&input[ref_i], NULL, 16);
    ref_o++;
    ref_i += 3;
  }
}

void set_key() {
  int fd;
  char keyname[MAXPATHLEN];

  if ((fgets(input, MAX_INPUT_SIZE, stdin)) == NULL){
    perror("fgets");
    return;
  }
  input[(strlen(input) - 1)] = '\0';  /* strip \n */
  get_home(keyname);
  strcat(keyname, PKE_PATH);
  strcat(keyname, input);
  strcat(keyname, ".pub");

  if ((fd = open(keyname, O_RDONLY)) < 0) {
    fprintf(stderr, "while trying to open %s \n", keyname);
    perror("open");
    return;
  }
  read(fd, &key.bits, sizeof(key.bits));
  read(fd, &key.modulus, sizeof(key.modulus));
  read(fd, &key.exponent, sizeof(key.exponent));

  key_good = 1;
}

void reset_structure() {
  /* This should reset the data structure to empty */
```

```
  pke_elt *next, *pos = head;

  if (pos != NULL)
    next = pos->next;
  while (pos != NULL){
    free(pos);
    pos = next;
    if (next != NULL)
      next = next->next;
  }
  head = NULL;
  current_num = 0;
}

void pad_with_random(char* string)
{ /* Pads the string with appropriate length random data */
  char *ptr = string, *end = (string + RANDOM_PAD_SIZE);

  srand(time(NULL));

  while (ptr < end)
    *ptr++ = (char) rand();
}


void create() {
  /* This function implements the Create: operation */
  /* Make sure we've cleared everything */
  pke_elt *ptr, *prev;
  char new_char;
  int done = 0;

  reset_structure();
  ptr = head;
  prev = head;

  /* We need a key value */
  if (!key_good) {
    fprintf(stderr, "No key supplied!");
    return;
  }

  /* No we read a number which is a decimal ASCII code for the byte
     that will go into the structure., Until we read 'End' we keep
```

```
     going */

  while (!done) {
    if ((fgets(input, MAX_INPUT_SIZE, stdin)) == NULL) {
      perror("fgets");
      exit(1);
    }
    if (!strncasecmp(input, "end", (sizeof("end") - 1)))
      done = 1;
    else {
      if ((ptr = (struct pke_elt_struct *)
   malloc(sizeof(struct pke_elt_struct))) == NULL) {
fprintf(stderr, "malloc failed.. out of memory?\n");
exit(1);
      }
      ptr->next = NULL;
      if (prev == NULL) {
head = ptr;
prev = head;
      }
      else {
prev->next = ptr;
prev = ptr;
      }

      /* Decode the byte into a data block with randomizer */
      new_char = (char)atoi(input);
      pad_with_random(input);
      input[RANDOM_PAD_SIZE] = new_char;
      input[RANDOM_PAD_SIZE + 1] = '\0';

      /* Now encrypt it */
      PKE_encrypt(input, (int)(DATA_LEN), ptr);

      /* update our 'cached value' */
      current_num++;
    }
  }
  /* make the cached value hold the right data */
  current_ptr = ptr;
}

void modify() {
  /* This function implements the Modify: operation */
```

```
  int cur_pos = 1, searching;
  pke_elt *cur_ptr = head;
  char newdata[DATA_LEN + 1];

  if ((fgets(input, MAX_INPUT_SIZE, stdin)) == NULL) {
    perror("fgets");
    return;
  }
  if ((sscanf(input, "%d", &searching)) != 1){
    fprintf(stderr, "sscanf failure");
    return;
  }
  if ((fgets(input, MAX_INPUT_SIZE, stdin)) == NULL) {
    perror("fgets");
    return;
  }
  input[(strlen(input) - 1)] = '\0';  /* strip \n */

  /* use cache if can */
  if ((current_num <= searching) &&
      (current_num != 0)) {
    cur_pos = current_num;
    cur_ptr = current_ptr;
  }
  while ((cur_pos != searching) &&
 (cur_ptr != NULL)){
    cur_ptr = cur_ptr->next;
    cur_pos++;
  }
  if ((cur_pos == searching) &&
      (cur_ptr != NULL)) {
    pad_with_random(newdata);
    decode(input, &newdata[RANDOM_PAD_SIZE]);
    PKE_encrypt(newdata, DATA_LEN, cur_ptr);
    current_num = cur_pos;
    current_ptr = cur_ptr;
  }
  else
    if (cur_ptr == NULL){
      fprintf(stderr, "modify ran past end of structure\n");
      return;
    }
    else {
      fprintf(stderr, "problems modifying\n");
      return;
```

```
    }
}

void delete() {
  /* This function implements the Delete: operation */

  int searching, cur_pos = 1;
  pke_elt *prev_ptr = NULL, *cur_ptr = head;

  if ((fgets(input, MAX_INPUT_SIZE, stdin)) == NULL) {
    perror("fgets");
    return;
  }
  if ((sscanf(input, "%d", &searching)) != 1){
    fprintf(stderr, "sscanf failure");
    return;
  }

  /* use cache if can */
  if ((current_num < searching) &&
      (current_num != 0)) {
    cur_pos = current_num + 1;
    cur_ptr = current_ptr -> next;
    prev_ptr = current_ptr;
  }

  while ((cur_pos != searching) &&
 (cur_ptr != NULL)){
    prev_ptr = cur_ptr;
    cur_ptr = cur_ptr->next;
    cur_pos++;
  }

  if ((cur_ptr != NULL) &&
      (cur_pos == searching))
    if (prev_ptr == NULL) {  /* we're deleting the first element */
      head = cur_ptr -> next;
      free(cur_ptr);
      current_num = 0;  /* blow away cache */
    }
    else{
      prev_ptr->next = cur_ptr->next;
      free(cur_ptr);
      current_num = cur_pos - 1;
      current_ptr = prev_ptr;
```

```
    }
  else{
    fprintf(stderr, "error deleting\n");
  }
}

void insert() {
  /* This function implements the Insert: operation */
  int searching, cur_pos = 1;
  pke_elt *prev_ptr = NULL, *cur_ptr = head, *temp;
  char newdata[DATA_LEN + 1];

  if ((fgets(input, MAX_INPUT_SIZE, stdin)) == NULL) {
    perror("fgets");
    return;
  }
  if ((sscanf(input, "%d", &searching)) != 1){
    fprintf(stderr, "sscanf failure");
    return;
  }
  if ((fgets(input, MAX_INPUT_SIZE, stdin)) == NULL) {
    perror("fgets");
    return;
  }
  input[(strlen(input) - 1)] = '\0';  /* strip \n */

  /* use cache if can */
  if ((current_num < searching) &&
      (current_num != 0)) {
    cur_pos = current_num + 1;
    cur_ptr = current_ptr -> next;
    prev_ptr = current_ptr;
  }

  while ((cur_pos != searching) &&
 (cur_ptr != NULL)){
    prev_ptr = cur_ptr;
    cur_ptr = cur_ptr->next;
    cur_pos++;
  }
  if (prev_ptr == NULL){  /* we're inserting at the head */
    if ((prev_ptr = (struct pke_elt_struct *)
 malloc(sizeof(struct pke_elt_struct))) == NULL){
      perror("malloc");
      exit(1);
```

```
    }
    pad_with_random(newdata);
    decode(input, &newdata[RANDOM_PAD_SIZE]);
    PKE_encrypt(newdata, DATA_LEN, prev_ptr);
    head = prev_ptr;
    prev_ptr->next = cur_ptr;
    current_num = 2;
    current_ptr = cur_ptr;
  }
  else {
    if (cur_pos == searching){
      if ((temp = (struct pke_elt_struct *)
   malloc(sizeof(struct pke_elt_struct))) == NULL){
perror("malloc");
exit(1);
      }
      pad_with_random(newdata);
      decode(input, &newdata[RANDOM_PAD_SIZE]);
      PKE_encrypt(newdata, DATA_LEN, temp);
      prev_ptr->next = temp;
      temp -> next = cur_ptr;
      current_num = searching;
      current_ptr = temp;
    }
    else {
      fprintf(stderr, "confused!!");
      exit(1);
    }
  }
}

void save_file() {
  /* This function implements the Save: operation */
  int fd;
  pke_elt *cur_ptr = head;

  if ((fgets(input, MAX_INPUT_SIZE, stdin)) == NULL) {
    perror("fgets");
    return;
  }
  input[(strlen(input) - 1)] = '\0';  /* strip \n */
  if ((fd = open(input, O_RDWR|O_CREAT, S_IRUSR | S_IWUSR |
 S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) <0) {
    perror("failed to open file for writing");
    exit(1);
```

```
  }

  while(cur_ptr) {
    if ((write(fd, &cur_ptr->datalen, sizeof(int))) < sizeof(int)){
      perror("couldn't write");
      return;
    }
    if ((write(fd, cur_ptr->data, cur_ptr->datalen)) <
        cur_ptr->datalen) {
      perror("couldn't write");
      return;
    }
    cur_ptr = cur_ptr -> next;
  }
  close(fd);
  printf("Done\n");
}

void load_file() {
  /* This function implements the Load: operation */
  int fd, done = 0;
  pke_elt *cur_ptr, *prev_ptr = NULL;

  reset_structure();

  if ((fgets(input, MAX_INPUT_SIZE, stdin)) == NULL) {
    perror("fgets");
    return;
  }
  input[(strlen(input) - 1)] = '\0';  /* strip \n */
  if ((fd = open(input, O_RDONLY)) <0) {
    perror("failed to open file");
    exit(1);
  }

  while (!done) {
    if ((cur_ptr = (struct pke_elt_struct *)
 malloc(sizeof(struct pke_elt_struct))) == NULL){
      fprintf(stderr, "malloc failed to find enough memory");
      exit(1);
    }
    if ((read(fd, cur_ptr->data, MAX_ENC_DATA_LEN)) <
MAX_ENC_DATA_LEN){
      done = 1;
      free(cur_ptr);
```

```
      if (prev_ptr == NULL)
head = NULL;
      else
prev_ptr -> next = NULL;
    }
    else {
      if (prev_ptr == NULL){
head = cur_ptr;
prev_ptr = cur_ptr;
      }
      else{
prev_ptr -> next = cur_ptr;
prev_ptr = cur_ptr;
      }
    }
  }
}

/* todo */
void PKE_encrypt (char * input, int inlen, pke_elt *output) {
  /* encrypt the string in input to the string in output...
     assumes output has enough space to store data */
  int status, outlen;

  if (!key_good){
    fprintf(stderr, "no key!!\n");
    exit(1);
  }
  if ((status = RSAPublicEncrypt
    (output->data, &outlen, input, inlen, &key, &randomStruct))
      != 0){
    fprintf(stderr, "Having problems encrypting");
    exit(1);
  }
  output->datalen = outlen;
}

void get_home(result)
char *result;
{
  struct passwd *pwd;

  if (strcpy(result,(char *)getenv("HOME")))
    return;
  if (pwd = getpwuid(getuid())){
```

```
    strcpy(result, pwd->pw_dir);
    return;
  }
  else{
    strcpy(result, "/");
    return;
  }
}

void Init_setup() { /* try to initialize things */
  static unsigned char seedByte = 0;
  unsigned int bytesNeeded =1000;
  struct timeval tp;

  setbuf(stdout, NULL);
  R_RandomInit (&randomStruct);

  gettimeofday(&tp);
  seedByte = (char)tp.tv_usec;
  R_RandomUpdate (&randomStruct, &seedByte, 1);

  while (bytesNeeded > 0) {
    gettimeofday(&tp);
    seedByte = (char)tp.tv_usec;
    R_RandomUpdate (&randomStruct, &seedByte, 1);
    R_GetRandomBytesNeeded (&bytesNeeded, &randomStruct);
  }
}
```

PKE.h

The associated header file.

```
/* This file describes all the constants and macros used in PKE.c. */

#include <global.h>
#include <rsaref.h>
#include <rsa.h>

#define RANDOM_PAD_SIZE 10     /* number of random bytes to use */
#define DATA_SIZE 1            /* number of bytes of data */
#define MAX_INPUT_SIZE 256     /* maximal length of a line */
```

```
#define PKE_PATH "/.pke_keys/" /* extension to $HOME where keys are */


/* Things that need no touching */

#define DATA_LEN (DATA_SIZE + RANDOM_PAD_SIZE)
          /* length in bytes of the .data component */
#define MAX_ENC_DATA_LEN MAX_RSA_MODULUS_LEN
          /* length of encrypted data block */

/* for the main loop */
#define INPUT_ERROR 0
#define INPUT_QUIT 1
#define INPUT_KEY 2
#define INPUT_CREATE 3
#define INPUT_INSERT 4
#define INPUT_MODIFY 5
#define INPUT_DELETE 6
#define INPUT_SAVE 7
#define INPUT_LOAD 8

typedef struct pke_elt_struct {
  int datalen;
  char data[MAX_ENC_DATA_LEN];
  struct pke_elt_struct *next;
} pke_elt;
```

A.3.3 PKE-keygen.c

This c code is responsible for generating valid keypairs for the PKE.el mode. It uses RSAREF extensively, and the random number generator it uses is probably not fully trustworthy. It relies on *PKE.h*.

```
/* Code to generate a key pair for a particular username
 *  Probably could be better and more random, but
 *  good enough for our purposes.
 */

#include "PKE.h"

#include <stdio.h>
```

```
#include <sys/param.h>
#include <unistd.h>
#include <pwd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <macros.h>
#include <sys/time.h>

#include <global.h>
#include <rsaref.h>
#include <nn.h>

static struct termios save_termios;
static int ttysavefd = -1;
static enum {RESET, CBREAK} ttystate = RESET;

void get_home(char *), tty_atexit(void),
  InitFile(char *, int *, int *),
  OutputKeys(R_RSA_PUBLIC_KEY, int, R_RSA_PRIVATE_KEY, int),
  InitRandomStruct (R_RANDOM_STRUCT *randomStruct),
  PrintError (char *, int);
int tty_cbreak(int), tty_reset(int);

int main(int argc, char *argv[]){
  int keySize, fpub, fpriv, status;
  R_RANDOM_STRUCT randomStruct;
  R_RSA_PUBLIC_KEY key_pub;
  R_RSA_PRIVATE_KEY key_priv;
  R_RSA_PROTO_KEY protoKey;
  char buffer[10];

  atexit(tty_atexit);
  setbuf(stdin, NULL);
  setbuf(stdout, NULL);

  if (argc != 2){
    fprintf(stderr, "usage %s {username}\n", argv[0]);
    exit(1);
  }
  InitRandomStruct(&randomStruct);
  InitFile(argv[1], &fpub, &fpriv);

  /* initiate the proto structure */
```

```
  keySize = 0;
  while((keySize <512) || (keySize > 1024)) {
    printf("Please select a keysize (512 - 1024) : ");
    fflush(NULL);
    fgets(buffer, 9, stdin);
    sscanf(buffer, "%d", &keySize);
  }
  protoKey.bits = (unsigned int)keySize;
  protoKey.useFermat4 =1;
  printf("Generating keys. This will take a while. Please wait...\n");
  if (status = R_GeneratePEMKeys
       (&key_pub, &key_priv, &protoKey, &randomStruct)) {
    PrintError ("Generating keys", status);
    exit(1);
  }
  printf("       done.  ... Saving keys\n");
  OutputKeys(key_pub, fpub, key_priv, fpriv);
  R_RandomFinal(&randomStruct);
  R_memset((POINTER)&key_priv, 0, sizeof(key_priv));
  return(0);
  printf("Thanks for waiting. \n");
  close (fpub);
  close (fpriv);
}


void InitFile(char *name, int *fpub, int *fpriv) {
  char keypath[MAXPATHLEN];  /* yeah yeah yeah.. constant length */
  char *pos;  /* where extension goes */

  get_home(keypath);
  strcat(keypath, PKE_PATH);
  /* Test whether there exists a directory for keys */
  if (access(keypath, F_OK)) { /* failed to find file */
    if (mkdir(keypath, S_IRWXU)) {
      perror("Failed to create directory:");
      exit(1);
    }
  }
  strcat(keypath, name);
  pos = (&keypath[strlen(keypath)]);

  strcpy (pos, ".pub");
  if ((*fpub = open(keypath, O_RDWR|O_CREAT, S_IRUSR|S_IWUSR)) <0) {
    perror("failed to create keyfile:");
```

```
    exit(1);
  }
  strcpy (pos, ".prv");
  if ((*fpriv = open(keypath, O_RDWR|O_CREAT, S_IRUSR|S_IWUSR)) <0) {
    perror("failed to create keyfile:");
    exit(1);
  }
  pos[0] = '\0';
  printf("\n  Thanks\nWill store in %s.pub and .prv\n", keypath);
}

void OutputKeys(R_RSA_PUBLIC_KEY key_pub, int fpub,
R_RSA_PRIVATE_KEY key_priv, int fpriv) {

  /* Public Key first */
  write(fpub, &key_pub.bits, sizeof(key_pub.bits));
  write(fpub, &key_pub.modulus, sizeof(key_pub.modulus));
  write(fpub, &key_pub.exponent, sizeof(key_pub.exponent));

  /* Private key now */
  write(fpriv, &key_priv.bits, sizeof(key_priv.bits));
  write(fpriv, &key_priv.modulus, sizeof(key_priv.modulus));
  write(fpriv, &key_priv.publicExponent,
sizeof(key_priv.publicExponent));
  write(fpriv, &key_priv.exponent, sizeof(key_priv.exponent));
  write(fpriv, &key_priv.prime[0], sizeof(key_priv.prime[0]));
  write(fpriv, &key_priv.prime[1], sizeof(key_priv.prime[1]));
  write(fpriv, &key_priv.primeExponent,
sizeof(key_priv.primeExponent));
  write(fpriv, &key_priv.coefficient, sizeof(key_priv.coefficient));

  /* Done writing everything */
}

void get_home(result)
char *result;
{
  struct passwd *pwd;

  if (strcpy(result,(char *)getenv("HOME")))
    return;
  if (pwd = getpwuid(getuid())){
    strcpy(result, pwd->pw_dir);
    return;
  }
```

```
  else{
    strcpy(result, "/");
    return;
  }
}

void InitRandomStruct (R_RANDOM_STRUCT *randomStruct)
{
  static unsigned char seedByte = 0;
  unsigned int bytesNeeded =1000;
  int i= 40;
  struct timeval tp;

  tty_cbreak(STDIN_FILENO);

  R_RandomInit (randomStruct);

  gettimeofday(&tp);
  seedByte = (char)tp.tv_usec;
  R_RandomUpdate (randomStruct, &seedByte, 1);

  printf("Please type in random chars until I tell you to quit\n");
  printf("to go   ");
  while (bytesNeeded > 0) {
    if (( read (STDIN_FILENO, &seedByte, 1)) != 1) {
      perror("read");
      tty_reset(STDIN_FILENO);
      exit(1);
    }
    R_RandomUpdate (randomStruct, &seedByte, 1);

    gettimeofday(&tp);
    seedByte = (char)tp.tv_usec;
    R_RandomUpdate (randomStruct, &seedByte, 1);
    R_GetRandomBytesNeeded (&bytesNeeded, randomStruct);
    if (bytesNeeded > 18)
      printf(".");
    else
      printf("%d", (bytesNeeded/2));
    i -= 1;
    if (i==0){
      printf("\n%5d - ", (bytesNeeded/2));
      i = 40;
    }
  }
```

```
  tty_reset(STDIN_FILENO);
}

int tty_cbreak(int fd)
{
  struct termios buf;

  if (tcgetattr(fd, &save_termios) <0)
    return (-1);

  buf= save_termios;

  buf.c_lflag &= ~(ECHO | ICANON);

  buf.c_cc[VMIN] = 1;
  buf.c_cc[VTIME] = 0;

  if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
    return (-1);
  ttystate = CBREAK;
  ttysavefd = fd;
  return(0);
}

int tty_reset(int fd)
{
  if (ttystate != CBREAK)
    return (0);
  if (tcsetattr(fd, TCSAFLUSH, &save_termios) <0)
    return (-1);
  ttystate = RESET;
  return (0);
}

void tty_atexit(void)
{
  if (ttysavefd >= 0)
    tty_reset(ttysavefd);
}

void PrintError (char *task, int type)
{
  char *typeString, buf[80];

  if (type == 0) {
```

```
    puts (task);
    return;
  }

  /* Convert the type to a string if it is recognized.
   */
  switch (type) {
  case RE_KEY:
    typeString = "Recovered DES key cannot decrypt encrypted content";
    break;
  case RE_LEN:
    typeString =
      "Encrypted key length or signature length is out of range";
    break;
  case RE_MODULUS_LEN:
    typeString = "Modulus length is out of range";
    break;
  case RE_PRIVATE_KEY:
    typeString = "Private key cannot encrypt.";
    break;
  case RE_PUBLIC_KEY:
    typeString = "Public key cannot encrypt.";
    break;
  case RE_SIGNATURE:
    typeString = "Signature is incorrect";
    break;

  default:
    sprintf (buf, "Code 0x%04x", type);
    typeString = buf;
  }

  printf ("ERROR: %s while %s\n", typeString, task);
  fflush (stdout);
}
```

A.3.4 PKE-decrypt.c

This c code is responsible for taking as input the user's name whose private key is to be used to decrypt with, and also an input and output file (which may be replaced with "-" to use stdin or stdout). It also relies on *PKE.h*, and uses RSAREF.

```
/* This file is responsible for decrypting an encrypted file. It
   takes the following arguments:

   PKE-decode {name} {source} {dest}

        {name}   : name of person whose private key is to be
           used.

{source} : source filename, or '-' for stdin.
{dest}   : destination filename, or '-' for stdout.
*/

#include "PKE.h"

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <pwd.h>

void read_key(char *, R_RSA_PRIVATE_KEY *),
  pke_decrypt(int, int, R_RSA_PRIVATE_KEY),
  get_home(char *);

int main(int argc, char *argv[]) {
  int fdin, fdout;
  R_RSA_PRIVATE_KEY key;

  if (argc != 4) {
    fprintf(stderr, "usage: %s {name} {source} {dest}\n", argv[0]);
    fprintf(stderr, "\t where {source} and {dest} may be '-' for\n");
    fprintf(stderr, "\t stdin or stdout respectively\n");
    exit(1);
  }

  if (!strcmp("-", argv[2]))
    fdin = STDIN_FILENO;
  else
    if ((fdin = open(argv[2], O_RDONLY)) < 0) {
      perror("input open");
      exit(1);
    }
  if (!strcmp("-", argv[3]))
```

```
    fdout = STDOUT_FILENO;
  else
    if ((fdout = open(argv[3], O_RDWR|O_CREAT, S_IRUSR|S_IWUSR))<0){
      perror("output open");
      exit(1);
    }

  read_key(argv[1], &key);
  pke_decrypt(fdin, fdout, key);
}

void read_key(char *name, R_RSA_PRIVATE_KEY *key){
  /* reads a key in from {name}.prv, and returns the key */
  int fd;
  char keyname[MAXPATHLEN];

  get_home(keyname);
  strcat(keyname, PKE_PATH);
  strcat(keyname, name);
  strcat(keyname, ".prv");

  if ((fd = open(keyname, O_RDONLY)) < 0) {
    perror("key input open failed");
    exit(1);
    }

  read(fd, &(key->bits), sizeof(key->bits));
  read(fd, &(key->modulus), sizeof(key->modulus));
  read(fd, &(key->publicExponent), sizeof(key->publicExponent));
  read(fd, &(key->exponent), sizeof(key->exponent));
  read(fd, &(key->prime[0]), sizeof(key->prime[0]));
  read(fd, &(key->prime[1]), sizeof(key->prime[1]));
  read(fd, &(key->primeExponent), sizeof(key->primeExponent));
  read(fd, &(key->coefficient), sizeof(key->coefficient));

  close(fd);
}


void pke_decrypt(int fdin, int fdout, R_RSA_PRIVATE_KEY key) {
  /* read from in, decode, output to out */
  char enc_data[MAX_ENC_DATA_LEN], out_data[DATA_SIZE],
    temp[DATA_LEN];
  int outlen, status, inlen;
```

```
  while ((read(fdin, &inlen, sizeof(int)) == sizeof(int)) &&
 (read(fdin, enc_data, inlen) == inlen)) {
    if ((status = RSAPrivateDecrypt
 (temp, &outlen, enc_data, inlen, &key)) != 0)
      fprintf(stderr, "Error decrypting.. continuing\n");
    pke_decode(temp, out_data);
    if ((write(fdout, out_data, DATA_SIZE)) != DATA_SIZE)
      fprintf(stderr, "error writing data.. continuing\n");
  }
}

pke_decode(char *input, char *output){
  /* converts an encoded string with randomizer to just string */
  char *inptr = (input + RANDOM_PAD_SIZE) , *outptr;
  int count = 0;

  for (outptr = output;
       outptr < (output + DATA_SIZE);
       outptr++)
    *outptr = *inptr++;
}


void get_home(result)
char *result;
{
  struct passwd *pwd;

  if (strcpy(result,(char *)getenv("HOME")))
    return;
  if (pwd = getpwuid(getuid())){
    strcpy(result, pwd->pw_dir);
    return;
  }
  else{
    strcpy(result, "/");
    return;
  }
}
```

Bibliography

[1] W. B. Alexi, B. Chor, O. Goldreich, and C. P. Schnorr. RSA/Rabin bits are $1/2+1/poly(\log(N))$ secure. In *Proc. 25th IEEE Symp. on Foundations of Comp. Science*, pages 449–457, Singer Island, 1984. IEEE.

[2] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: the case of hashing and signing. In Yvo G. Desmedt, editor, *Proc. CRYPTO 94*, pages 216–233. Springer, 1994. Lecture Notes in Computer Science No. 839.

[3] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography with application to virus protection. In *Proc. 27th ACM Symp. on Theory of Computing*, pages 45–56, Las Vegas, 1995. ACM.

[4] Mihir Bellare and Shafi Goldwasser. Lecture notes on cryptography. for summer course in cryptography, 1996.

[5] Mihir Bellare, Roch Guérin, and Phillip Rogaway. XOR MACs: New methods for message authentication using block ciphers. Technical Report RC 19970, IBM Research Report, March 1995.

[6] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In Yvo G. Desmedt, editor, *Proc. CRYPTO 94*, pages 341–358. Springer, 1994. Lecture Notes in Computer Science No. 839.

[7] Matt Blaze and Martin Strauss. Proxy cryptography. Draft made on May 13th, 1997, 1997.

[8] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, November 1976.

[9] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-malleable cryptography. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 542–552. ACM, 1991.

[10] S. Goldwasser and S. Micali. Probabilistic encryption. *JCSS*, 28(2):270–299, April 1984.

[11] Frans Kaashoek. 6.033 lecture notes. lecture notes for computer systems engineering course, 1996.

[12] Daniele Micciancio. Oblivious data structures: Applications to cryptography. In *Proc. 29th ACM Symp. on Theory of Computing*, 1997.

[13] Rafail Ostrovsky. Efficient computations on oblivious rams. In *Proc. 22nd ACM Symp. on Theory of Computing*, Baltimore, 1990. ACM.

[14] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[15] RSA Laboratories. *RSAREF(TM): A Cryptographic Toolkit, Library Reference Manual*, 2.0 edition, March 1994.

[16] Bruce Schneier. *Applied Cryptography Second Edition: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, New York, 1996.

[17] J.G. Steiner, B.C. Neuman, and J.I. Schiller. Kerberos: an authentication service for open network systems. In *Usenix Conference Proceedings*, pages 191–202, Dallas, Texas, February 1988.

[18] Philip Zimmerman. *PGP(tm) User's Guide.* Phil's Pretty Good Software, 1994.