Practical Implementation and Analysis of Hyper-Encryption

by

Jason K. Juang

S.B., C.S., M.I.T., 2008

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2009

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by . . . .
Ronald L. Rivest
Viterbi Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . .
Michael O. Rabin
T.J. Watson Sr. Professor of Computer Science, Harvard SEAS
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Practical Implementation and Analysis of Hyper-Encryption

by

Jason K. Juang

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The security of modern cryptographic schemes relies on limits on computational power and assumptions about the difficulty of certain mathematical problems, such as integer factorization. This thesis describes *hyper-encryption in the limited access model*, a system that provides perfect secrecy and authentication against an adversary who possesses unbounded computational power, but who is limited in the ability to eavesdrop on more than a fraction of computers in a large network, such as the Internet. This thesis also presents an implementation of hyper-encryption in the limited access model, discusses areas where the theoretical system differs from the practical implementation, and analyzes their impact on the security of the system.

Thesis Supervisor: Ronald L. Rivest
Title: Viterbi Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Michael O. Rabin
Title: T.J. Watson Sr. Professor of Computer Science, Harvard SEAS

Acknowledgments

This thesis would not have been possible without the support and aid of many people, including but by no means limited to:

Professors Michael Rabin and Ron Rivest, for their invaluable guidance and feedback throughout the project and the thesis-writing process.

Harvard students Yan Cheng Chang, Mike Hamburg, Cassia Martin, Bryan Parno, Alex Rampell, Mike Schnall-Levin, L. Storzek, Kartik Venkatram, David Xiao, and possibly others, whom I have never met, but whose previous implementation work on hyper-encryption became the basis for this project.

Jeff Perkins and Professor Michael Ernst, whose advice and guidance during my first research experience at MIT helped me gain footing in uncertain territory.

Professors Ron Rivest, Shafi Goldwasser, and Sivan Toledo, for taking me on as TA for their classes and tolerating the frequent, verbose, and nagging e-mails that ensued.

Harvey Jones, for working with me on the 6.857 project that got me interested in security in the first place, and for his advice and friendship over the intervening years. (We finish each other's. . . sandwiches.)

Finally, my family: my parents and my brother Phil, for their years of love and support, for believing in me, for encouraging me to keep going when I felt like giving up, and for pushing me to achieve to the best of my ability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The security of most modern cryptographic schemes relies on limits on computational power. Encrypted messages sent today may not remain secret in the future if the encryption key becomes known, if the cryptographic scheme proves mathematically weak, or when technology improves to the point where brute-force decryption becomes possible. An eavesdropper who records encrypted transmissions may be able to decrypt them and read sensitive information at some point in the future.

For example, the confidentiality of messages encrypted using RSA [1] depends on the assumption that factoring integers is hard. The confidentiality of messages encrypted using ElGamal [2] encryption depends on the assumption that computing discrete logarithms is hard. If algorithms were to be found that efficiently factor integers or calculate discrete logs in large finite groups, these algorithms could be used to efficiently decrypt messages encrypted with RSA or with ElGamal. Furthermore, quantum computers, if ever practically built, will be able to factor large integers and compute discrete logs, thus, in principle, posing a (future) threat to these encryption methods.

In order to securely send messages that need to remain secret *forever*, we must develop a practical cryptographic system that provides secrecy guarantees against an adversary with *unbounded* computational power.

Hyper-encryption is one such system described by Rabin [3]. Hyper-encryption employs a multitude of servers (*Page Server Nodes*, or *PSNs*) from which anyone may request a page of random bits. Each page is served at most twice before being destroyed and replaced with a new random page. Thus, two parties may construct a shared one-time pad by using a short initial shared secret key, on the order of a few thousand bits, and using those random bits to download the same set of pages from the network of PSNs.

Because each PSN destroys any page that has been requested twice, the one-time pad remains secret even if the initial shared key is later divulged; the pages that comprise the pad are no longer available via the PSN network, so reissuing page requests using the original key will yield different pages.

Hyper-encryption provides benefits over the straightforward one-time pad scheme, where two parties meet and exchange a very large one-time pad (for example, a portable hard disk full of random data). Hyper-encryption requires a relatively short shared secret, on the order of a couple kilobytes. After the initial secret is established, it enables the parties to create new one-time pads when needed, thus enabling them to communicate indefinitely while keeping only a few megabytes of state.

This thesis describes an implementation of hyper-encryption in Java. It presents implementations for the Page Server Nodes, and for client software that enables users to send hyper-encrypted messages via e-mail. It discusses vulnerabilities, including denial-of-service vulnerabilities, that result both from weaknesses in the original protocols, and from design decisions made in the implementation. It describes the consequences of these vulnerabilities, and presents strategies for minimizing their impact.

The rest of this document is organized as follows. Chapter 2 discusses relevant background and work related to hyper-encryption. Chapter 3 describes the theory and abstract system model behind hyper-encryption, including various improvements over previously described protocols. It suggests possible implementation strategies, evaluates threats and vulnerabilities posed by the suggested implementations, and analyzes their impact. Chapter 4 presents the design of our implementation. This includes a description of how the abstract elements of the hyper-encryption model correspond to the concrete elements of our implementation. Chapter 5 recommends specific tasks for further improvement, as well as direction for future work related to our implementation and to hyper-encryption in general. Finally, Chapter 6 summarizes the contributions of this work.

Chapter 2

Background and Related Work

2.1 True random number generation

Many cryptographic applications call for truly random bits. For applications (such as ours) that require a uniform random source of bits that is unpredictable by any adversary, pseudorandom number generators (PRNG) are unsuitable.

Simple methods of generating random numbers include flipping coins and rolling dice. However, these methods tend to be inefficient, and very tedious for the person tasked with rolling several thousand dice and recording the results.

A more manageable method is to extract randomness from a physical phenomenon. There are free services online that provide random numbers generated from atmospheric noise[1] and radioactive decay[2]. The LavaRnd service[3] extracts random numbers from the CCD chip of a webcam; it is the spiritual successor to *lavarand*, which used a camera pointed at a lava lamp. There is a variety of techniques for extracting randomness from such weak random sources; Shaltiel [4] provides a survey of this field.

[1]`http://www.random.org/`
[2]`http://www.fourmilab.ch/hotbits/`
[3]`http://www.lavarnd.org/`

2.2 Rabin fingerprints

Rabin [3] [5] describes an efficient, provably secure authentication mechanism that uses irreducible polynomials to compute the *fingerprint* of a message.. His method uses a p-bit shared secret key w to create a random irreducible polynomial g_w of prime degree p. The message M is then expressed as a polynomial, and the *fingerprint* is the residue when this polynomial is divided by g_w. He shows that an adversary can forge a message and accompanying fingerprint with probability no greater than $m \cdot 2^{-p}$, where m is the length of the plaintext, in bits, and that the probability can be made arbitrary small by choosing a larger p or by using multiple such random polynomials.

This method is very efficient: if the random irreducible polynomials are generated and saved offline, then computing the fingerprint of a message requires only m XOR operations.

2.3 One-time pad

One well-studied method of ensuring confidentiality against an adversary with unbounded computational power is the one-time pad (OTP). Suppose two parties, Alice and Bob, are communicating, and Alice wants to send an m-bit long message to Bob. Alice and Bob choose, in advance, a secret string of m independent random bits to be used to encrypt this message. To encrypt, Alice computes the XOR of her message M and the pad S, and transmits the ciphertext $C = M \oplus S$. To decrypt C, Bob computes the plaintext $M = C \oplus S$.

Claude Shannon [6] showed that OTP is information-theoretically secure, even if the adversary has some prior information about the message. Let $P(M)$ be the *a priori* probability that the message is M. Then $P(M \mid C)$ is the *a posteriori* probability of the message M; it is the probability of M, given the ciphertext $C = M \oplus S$. We will show that $P(M \mid C) = P(M)$: the adversary gains no information from the ciphertext C.

Because the pad S consists of m independent random bits, $P(S) = 2^{-m}$. From the standpoint of the adversary, the probability of observing a ciphertext C, given the plaintext M, is $P(C \mid M) = P(S = (C \oplus M)) = 2^{-m}$.

Thus, the probability of observing a particular ciphertext C is

$$\begin{aligned} P(C) &= \sum_M P(C \mid M) \cdot P(M) \\ &= \sum_M 2^{-m} \cdot P(M) \\ &= 2^{-m} \end{aligned}$$

We can now apply Bayes' Rule to find the *a posteriori* probability of M:

$$\begin{aligned} P(M \mid C) &= \frac{P(C \mid M) \cdot P(M)}{P(C)} \\ &= \frac{2^{-m} \cdot P(M)}{2^{-m}} \\ &= P(M) \end{aligned}$$

Thus, the adversary gains no information about M from seeing C. This result holds in the absence of any assumptions about the adversary's computing power; the adversary gains no information even if his computing power is unbounded.

It is crucial that this "pad" of bits only ever be used *once* (hence, "one-time pad"). If a pad is used to encrypt several different messages $C_i = M_i \oplus S$, then an eavesdropper can gain information about the messages by computing $C_i \oplus C_j = (M_i \oplus S) \oplus (M_j \oplus S) = M_i \oplus M_j$. This may allow the attacker to recover information about M_i and M_j, and in some cases, recover portions of the pad, thus revealing substantial portions of all the messages M_i.

One major hurdle to use of OTP is that, in general, encrypting an m-bit message requires an m-bit long shared secret key. Specifically, Shannon [6] proves that for *any* cipher, if the adversary and the legitimate users have access to exactly the same information, apart from the secret key, then “the amount of uncertainty we can introduce into the solution cannot be greater than the key uncertainty.”

Section 2.4 discusses several approaches to relaxing Shannon’s assumption that the adversary has access to all the same information as the users.

2.4 Provably secure schemes

Previously proposed provably secure cryptographic schemes have worked with a variety of assumptions that limit the adversary’s information, but not his computational power.

2.4.1 Quantum cryptography

One approach to generating a shared secret in the presence of an adversary uses the principles of quantum mechanics to guarantee secrecy.

In 1984, Bennett and Brassard [7] demonstrated a protocol for key distribution, now known as *BB84*, that exploits the uncertainty inherent in measuring polarized photons. Alice and Bob can generate a shared, secret one-time pad through use of a quantum channel, such as a private fiber optic line, followed by public discussion over a classical communication channel. Furthermore, BB84 allows Alice and Bob to detect a third party eavesdropping on the quantum channel with extremely high probability.

Other protocols for quantum key exchange have since been invented, but they share many of the same disadvantages. Namely, the equipment required (e.g., a machine capable of transmitting a single photon with a specified polarization) can be prohibitively expensive.

2.4.2 Noisy channel model

Maurer [8] describes a key exchange protocol where two parties, Alice and Bob, and an eavesdropper, Eve, all receive the output of a common source through independent binary symmetric channels (channels that randomly introduce bit errors). Alice and Bob each have information not available to the adversary: the error introduced by their channels. Alice and Bob can now generate a secret key through public discussion, despite Eve's partial knowledge of their original information.

2.4.3 Bounded storage model

The bounded storage model, as formulated by Maurer [9], places limits on the amount of storage available to an adversary, but *not* on the space or time available for *computation*. That is, the adversary is computationally unbounded, but has some finite, although potentially large, amount of space for storage.

Maurer [9] describes a cipher that requires the existence of a large public random string α. This cipher provably provides perfect secrecy under the assumption that the adversary may only access and store some fraction of α. More precisely, the adversary gains no information about the message unless he examines a "substantial fraction" of α.

Cachin and Maurer [10], and Aumann and Rabin [11], in a stronger result, show information-theoretically secure schemes under a stronger assumption. The adversary may access any or all of α and may "use unlimited computing power to compute any probabilistic function" of α, so long as the *output* of the function fits in his limited available memory.

It is not clear that the assumptions of the bounded storage model are valid, or if they are, that they will continue to hold. The schemes cited above suppose the availability of a high-bandwidth public source, such as a satellite that generates and broadcasts random bits. They also rely upon the assumption that it is prohibitively expensive for an adversary to capture and record all of the broadcast bits. Storage is cheap, with commercially-available hard disks costing well under $1 per gigabyte, and prices falling rapidly as technology improves. At the same time, bandwidth is falling in cost, but it is not clear that improvements in bandwidth are outpacing the growth of affordable storage capacity. Furthermore, it is much cheaper to upgrade storage capacity on the ground than it is to upgrade the bandwidth of one or more satellites.

2.4.4 Limited access model

Like the bounded storage model, Rabin's limited access model [3] postulates a computationally unbounded adversary and a method of obtaining public random bits. However, rather than a single source of random bits, the bits are distributed across a multitude of *Page Server Nodes* (PSNs), each of which contains and continually replenishes a collection of pages of random bits. Each PSN serves a page, in response to requests, *at most twice* before destroying it.

The limited access model supposes that it is impossible for an adversary to monitor more than some fraction of accesses made by a user, Alice, to the network of PSNs. This implies both that the eavesdropper is unable to read *all* of Alice's communications, and that the adversary cannot compromise *all* the PSNs.

Ways in which the adversary could compromise a PSN include gaining access to its store of random pages, monitoring its network traffic, or modifying it in some malicious way. The limited access model postulates that it is infeasible for an adversary to compromise more than some fraction (e.g., one half) of all PSNs. This is a reasonable assumption in the case where we have several thousand PSNs distributed all over the world.

Under this model, an adversary with unbounded storage gains nothing by requesting and storing all available public random bits. Any obtained bits can only ever be obtained by one other party before they are destroyed by the PSNs, and as a result, they will never be used as part of a shared secret.

Rabin [3] describes hyper-encryption in the limited access model, the focus of this thesis, in a 2005 paper. Alice and Bob share an initial secret key K. They use K to select random PSNs and request pages from those PSNs. They then reconcile their pages into common one-time pads. They use these pads to communicate securely, as well as to extend K to allow for future communication.

The limited access model as used by hyper-encryption supposes that the adversary is limited in his ability to eavesdrop on communications between the user (Alice) and the PSNs. It is worth considering a similar-sounding but distinct assumption: that the adversary is unable to eavesdrop on all communications between Alice and Bob. This is similar to Maurer's noisy channel model [8], as mentioned in Section 2.4.2. Alice and Bob can establish some common data, about which Eve has imperfect information due to the limit on her ability to eavesdrop. They can then extract a secret key through public discussion.

Chapter 3 of this thesis presents refinements of the original hyper-encryption protocols, including a more specific discussion of the limited access model in Section 3.3.1. Chapter 4 describes a working implementation of the system, suggesting that hyper-encryption can be used in practice.

Chapter 3

Theoretical System Model and Protocols

Hyper-encryption is a system for allowing two parties, starting with a shared secret key, to make use of a network of publicly-available computers (*page server nodes*) that provide pages of random data. The parties can use these pages to create a one-time pad of arbitrary length. This one-time pad can be used for authenticated, information-theoretically secure communication between the parties. The pad can be extended when needed by using a portion of the pad to create a new pad, in a manner similar to the first.

This chapter provides a specification for the hyper-encryption system and associated protocols. For the remainder of the chapter, let us consider a pair of partners, Alice and Bob (A and B); Alice encrypts messages and sends them to Bob, who decrypts them.

The material presented in this chapter is largely adapted from Rabin's 2005 paper [3], although with some changes and additions.

3.1 Terminology

In a hyper-encryption system, there are a number of *users*. Two users who are communicating with each other using hyper-encryption are *partners*; Alice and Bob are an example of a pair of partners. Each user has one or more partners.

There is a large number of *page server nodes* (PSNs). A PSN provides pages of random data for any user to download. We define N_P to be the size of a page, in bytes. Each PSN has a collection of *pages*, each containing N_P bytes of random data. Section 3.4 contains details of the PSN operations.

Each user employs a *client*, a software system responsible for managing the one-time pad, as well as for carrying out the encryption and decryption of messages. The structure and operation of the client is described in further detail in Section 3.5.

3.2 Hyper-encryption overview

Alice and Bob create an initial one-time pad by agreeing on a shared secret of several thousand bytes. Once Alice and Bob have established this initial common one-time pad, they can encrypt messages with it, and use it to authenticate messages they send to each other. They can make this shared pad longer by using some of it to download new pages of random data from the network of PSNs. This allows them to create an arbitrarily long one-time pad, and thus carry on secure communication in perpetuity.

Downloading a page requires using up some of the one-time pad. The client uses a pair of words from Alice and Bob's pad to choose a PSN and request a page from that PSN. This process is detailed in Section 3.5.4.

After downloading pages from the PSNs, Alice and Bob's clients make sure they have the same set of pages using *page reconciliation* (Section 3.5.5). In this protocol, Alice and Bob each compute fingerprints of the pages they have downloaded. Alice sends her list to Bob, who compares it with his list, and replies to indicate which fingerprints are common to both their lists. This allows them to determine which pages they have both downloaded; they use these common (secret) pages to add data to their one-time pad (*extending* the pad).

A part of the one-time pad is always reserved for the purpose of extending the pad in this manner; the remainder of the pad is used for encrypting and authenticating messages. Alice encrypts messages by XORing the plaintext against a part of the one-time pad (Section 3.5.7). She also computes a message authentication code (MAC) for the message, using a part of the pad as the secret key for a MAC based on Rabin's fingerprinting scheme (Section 3.5.8). Bob verifies the MAC, then decrypts the message by XORing the ciphertext against the same part of the pad. In either case, the used parts of the pad are discarded to ensure that they cannot be used again.

3.3 Adversarial model

This section models the adversary used in our analysis of hyper-encryption.

3.3.1 Capabilities

Computational power

The adversary has unbounded computational power, with one exception. We suppose that the adversary's computation power is great enough that he can break cryptographic schemes that rely on computational assumptions, but doing so requires a few weeks. For example, the adversary can factor integers with sufficient speed so that he may read any message encrypted using RSA within a few weeks.

Under this assumption, messages sent by Alice and Bob must be encrypted in an information-theoretically secure way in order to remain confidential forever. However, some hyper-encryption protocol messages only need to stay confidential for a few days; these protocols may employ schemes that rely upon computational assumptions. Specifically, we make this assumption to simplify the problem of establishing an initial shared secret, as discussed in Section 3.5.3. It allows users to use Diffie-Hellman or a similar, computationally secure, key exchange protocol to establish a (temporary) secret key, rather than a more cumbersome information-theoretically secure protocol.

Man-in-the-middle attacks

Alice and Bob exchange messages using some medium, such as a socket connection or e-mail messages. The adversary may actively attempt to modify these communications: he may modify messages in transit, and he may reorder messages.

Limited access to PSNs

The limited access model is introduced in Section 2.4.4. That section also discusses variants on the limited access model assumptions. This section focuses on the limited access model as we apply it to hyper-encryption.

The *limited access model assumption* is that the adversary can only eavesdrop on a fraction of all of Alice and Bob's PSN accesses. Example ways in which the adversary can eavesdrop on an access include gaining access to the PSN's store of random pages, monitoring the connection between Alice/Bob and the PSN, or modifying the PSN in some malicious way.

In a widely-deployed hyper-encryption system with thousands of PSNs operated by volunteers distributed throughout the internet, this seems to be a reasonable assumption. For the remainder of this chapter, we assume that the adversary can compromise no more than 1/5 of PSN accesses. That is, on 4/5 of all PSN accesses, the PSN operates correctly, and the adversary cannot read or modify the communications of, or gain any special access to, the PSN.

A related assumption of the limited access model is that the adversary cannot monitor all of Alice or Bob's communications with the PSNs. If the adversary can see all of Alice's communications, wherever she is, then clearly the adversary can construct the same one-time pad as Alice and Bob.

Were Alice to always access PSNs and perform hyper-encryption from, for example, her desktop computer in her office, monitoring her traffic would be trivial. Alice could evade such a wiretap by visiting a local coffee shop or other free public wireless access point, and downloading her pages from the PSN network from there. If she is afraid her personal computer is compromised, she may even download pages using a public computer or a friend's computer. Our implementation makes it straightforward to download pages on the road in this fashion.

If she is *really* paranoid, she may make arrangements with several trusted friends to each visit separate, randomly-chosen coffee shops, and each perform some fraction of the downloads on her behalf.

Clearly, there are situations in which the limited access model does not hold. For example, in a country in which all internet communications are monitored by the government, the government can subvert the limited access model. Similarly, if one internet service provider monopolizes an area, the provider can easily violate our assumption. However, in general, the assumption is a reasonable model.

While the limited access model assumption restricts the adversary's power to *compromise* PSN accesses, the adversary may still query any PSN in the normal way, i.e., by issuing a page request. He may issue as many legitimate page requests as he wishes.

3.3.2 Goals

We evaluate the security of hyper-encryption in terms of the secrecy it provides, and its resistance against forgery. We define these two properties in terms of games between an adversary and the partners Alice and Bob. In these games, all parties have a common list of n PSNs; the PSN list is provided by a trusted third party. The adversary's computational power is unbounded. Alice and Bob have a k-bit shared secret K drawn uniformly at random from $\{0,1\}^k$.

Secrecy

One goal of hyper-encryption is to keep messages secret. The adversary can defeat the system by gaining any information about a message sent from Alice to Bob. Suppose Alice, Bob, and the adversary play the following game:

1. The adversary chooses a sequence of bits $a_1, a_2, \ldots$, such that $\sum_1^k a_i \leq 1/5$, for all k. This sequence corresponds to which of Alice's PSN accesses on which the adversary may eavesdrop; at any given time, the adversary may only have eavesdropped on at most 1/5 of Alice's accesses. The adversary chooses a similar sequence, $b_1, b_2, \ldots$, for Bob. These sequences are known only to the adversary.
2. Alice and Bob use the bits of K to request pages from the PSNs and reconcile them, in order to establish 30 common pages with which to construct a common secret, S. If this is Alice's ith request, and $a_i = 1$, the adversary may choose the page. (Similarly for Bob and b_i.) Otherwise, the PSN supplies a random page known only to the requester.
3. The secret K is revealed to the adversary. Alice and Bob take the first k bits of S to create a new secret K for use in the next round.
4. The adversary sends to Alice two plaintexts, M_0 and M_1.
5. Alice randomly chooses a bit $b \in \{0, 1\}$ and publishes the hyper-encrypted message $C_b = E(M_b) = M_b \oplus P$.
6. The adversary attempts to determine the value of b.

The adversary wins if he correctly guesses the value of b. Otherwise, Alice and Bob win.

Alice, Bob, and the adversary repeat steps 2–6 of the game indefinitely; we refer to each iteration of the five steps as a *round.* If there is no round that the adversary can win with probability significantly greater than 1/2, then hyper-encryption ensures secrecy. That is to say, there is only a negligible probability that the adversary can gain any information about any message Alice and Bob send using hyper-encryption.

Unforgeability

Another way the adversary may defeat the system is by changing a message so that Bob receives a different message than the one that Alice sent. To this end, all messages, including protocol messages, must be authenticated. This is modeled by the following game, whose first three steps are the same as the secrecy game above:

1. The adversary chooses a sequence of bits $a_1, a_2, \ldots$, such that $\sum_1^k a_i \leq 1/5$, for all k. This sequence corresponds to which of Alice's PSN accesses on which the adversary may eavesdrop; at any given time, the adversary may only have eavesdropped on at most 1/5 of Alice's accesses. The adversary chooses a similar sequence, $b_1, b_2, \ldots$, for Bob. These sequences are known only to the adversary.
2. Alice and Bob use the bits of K to request pages from the PSNs and reconcile them, in order to establish 30 common pages with which to construct a common secret, S. If this is Alice's ith request, and $a_i = 1$, the adversary may choose the page. (Similarly for Bob and b_i.) Otherwise, the PSN supplies a random page known only to the requester.
3. The secret K is revealed to the adversary. Alice and Bob take the first k bits of S to create a new secret K for use in the next round.
4. The adversary sends Alice a plaintext M.
5. Alice sends the adversary the encryption of M, and corresponding MAC: $C = (E(M), MAC_M)$.
6. The adversary sends Bob some tuple $C' = (X, Y)$. Bob decides whether to *accept* or *reject* this tuple.

The adversary wins if he can find some $X \neq E(M)$ and some Y such that Bob accepts C', or if he can cause Bob to reject when $C' = C$. Otherwise, Alice and Bob win. That is, the adversary can defeat the system by either forging a message different from Alice's original message, or by denying Bob the ability to receive Alice's legitimate message.

Alice, Bob, and the adversary repeat steps 2–6 of the game indefinitely; we refer to each iteration of the five steps as a *round*. If there is no round that the adversary can win with non-negligible probability, then we consider hyper-encryption to provide protection against forgery.

3.4 The Page Server Node

3.4.1 Specification

A page server node (PSN) is a database of random pages. Clients may query a PSN and request a page of random bytes; each page consists of N_P bytes. A client U requests a page from PSN_j by sending it a word u, called the *request key*. The PSN replies by sending U a page from its database.

The precise manner by which the request key is used to retrieve a page from the database is intentionally left unspecified; Section 3.4.2 presents two possible implementations, and discusses their relative strengths and weaknesses. The only two requirements of the PSN are that

1. no page may be served more than two times before being destroyed; and
2. if two requests with identical request keys are both made within some reasonable time frame, they should result in the same page, with high probability.

Each PSN serves any given page of random data *at most twice*: upon the second request, the page must be destroyed. Thus, only two people have access to the page; if the two people who requested the page are Alice and Bob, then they can use the page for their one-time pad, knowing that nobody else has been served that page.

It is still possible that a PSN is malicious or malfunctioning, or that an attacker is eavesdropping on a PSN. To protect against these possibilities, Alice and Bob *reconcile* their pages (Section 3.5.5) to establish a common ordered list of pages. Alice and Bob then take their downloaded pages and XOR them together in groups, and append the resulting data to their one-time pad. Thus, an attacker must know *all* of the constituent pages in order to gain *any* information about the one-time pad. Under the limited access model assumption, the probability that the attacker has compromised all the PSNs involved is infinitesimal.

The definition of “reasonable time frame”, above, is unspecified, but we feel that something on the order of three days is a good guideline. Choosing a time frame requires considering a trade-off between convenience and efficiency. Two communicating parties should be able to obtain sets of common pages by using identical request keys, but without necessarily agreeing on a specific schedule for doing so. A longer time frame makes it more likely that two such parties will receive the same page, but may also require that the PSN keep track of more state; a shorter time frame allows the PSN more flexibility and may allow a more efficient implementation.

The motivation for specifying the “reasonable time frame” condition is to allow the PSN to (optionally) discard any “stale” pages on a regular basis. If a page is requested once, but a second request does not arrive for a long time (several days or weeks), then the PSN may assume that it will never arrive, and simply discard the key and page to save memory.

3.4.2 Possible implementations

Twice-Told Pad (2TP)

In this implementation, the PSN maintains two structures: *the queue* and *the map*. The queue is a FIFO queue of random pages that have never been served to a client. Newly-generated pages are appended to the queue. The map is a collection of pages $p_1, p_2, \ldots, p_k$ that have been requested exactly once, as well as the request key $w_1, w_2, \ldots, w_k$ associated with each page.

Upon receiving a request from user U with request key u, the PSN checks if u is in the list $w_1, w_2, \ldots, w_k$. If there exists some i such that $u = w_i$, then this is the second request with the supplied request key. The PSN removes w_i and p_i from the map, and responds to the request by sending the page p_i to U. The PSN then destroys p_i and w_i to ensure that p_i cannot be served a third or subsequent time; a third request with the key w_i will yield a different page.

If there does not exist any i such that $u = w_i$, then this request is the first request with the supplied request key. The PSN removes the first page from the queue and associates it with u. That is to say, it sets p_{k+1} to a new page from the queue, sets $w_{k+1} = u$, and adds the pair (w_{k+1}, p_{k+1}) to the map. It then sends p_{k+1} to U.

These procedures ensure that no page is "told" to clients more than twice, as required by the specification for a PSN.

A 2TP PSN may opt to impose a memory limit, keeping only some limited number of pages in the map. When a request causes this number to be exceeded, the 2TP evicts a page from the map; either a random page, or the least-recently requested. The 2TP may also opt to periodically go through the map and evict any stale pages.

L1

This implementation is named *L1* for its loose resemblance to a CPU cache. The PSN keeps a buffer P of k pages, and an access count $A[k]$ for each k. The buffer is initially filled with random pages, all with access counts of 0.

When a request arrives with key u, the PSN returns the page $P[u \bmod k]$, and increments its access count. If $A[u \bmod k]$ is now 2, the PSN discards $P[u \bmod k]$ and replaces it with a new random page, resetting the access count to 0. Similarly to the 2TP, these new random pages may be drawn from a FIFO queue of pre-generated pages; the queue is simply refilled when necessary and convenient.

The L1 implementation need not evict pages, as the 2TP does. Because every request matches one of the existing pages in the page buffer, the size to which the database grows is limited by k, and is entirely under control of the PSN.

Advantages and disadvantages

The L1 implementation has a slight performance advantage over 2TP. The 2TP PSN can implement $O(1)$ expected lookup times through use of a hash table; the L1 PSN provides $O(1)$ guaranteed time while using much less memory (only kN_P bytes).

Both L1 and 2TP are vulnerable to denial of service (DoS) attacks. An attacker can flood the PSN with random requests; in a 2TP PSN, this forces legitimately-requested pages to be evicted, while in a L1 PSN, this causes existing pages to be used up and replaced. In either case, the result is that legitimate users are less likely to get common pages.

2TP offers slightly more resistance against DoS. For L1, issuing k requests, with keys $0, 1, \ldots, k-1$, is sufficient to evict all the existing pages in the buffer, and thus effectively deny Alice and Bob a common page. For a 2TP PSN using a random eviction policy, it is more difficult to completely deny service: a given page is evicted with probability $\frac{1}{k}$, so after k requests from an attacker, the probability is $\left(\frac{k-1}{k}\right)^k$ that the page is still available. For $k = 1000$, this is approximately 36%.

In any event, DoS attacks are not a huge concern, as page reconciliation solves the case where users get different pages, and if there are sufficiently many PSNs, then a network-wide DoS attack is difficult.

3.5 The Client

The hyper-encryption client is the software, running on a user U's computer, that is responsible for managing the one-time pad needed for communication between U and each of U's partners. The client performs four operations, summarized towards the end of this chapter in Table 3.1: downloading pages, reconciling pages, creating blocks from pages, and encrypting/decrypting messages. These operations are described later in this section.

We only describe one direction of communication here (that is, Alice sending to

Bob). If Bob wants to send messages to Alice, they can use a similar setup for the other direction. This means that two partners A and B engaged in two-way communication have two completely independent one-time pads, managed independently: one used for encrypting and authenticating messages from A to B, and one used for encrypting and authenticating messages from B to A.

3.5.1 Encryption blocks and system blocks

The client divides up Alice and Bob's shared one-time pad into *blocks*. There are two types of blocks: *encryption blocks* and *system blocks*. Each block is assigned an integer *ID*. All system block IDs assigned over the lifetime of Alice and Bob's communication are distinct, as are all encryption block IDs. (It is permitted for a system block and an encryption block to share the same ID; this does not imply any relationship between the blocks.)

Encryption blocks are the part of the one-time pad used for encrypting and decrypting messages. Define N_E to be the size of each encryption block, in bytes.

System blocks are used for requesting and reconciling pages, and maintaining other metadata. Define N_S to be the size of each system block, in bytes. Each system block S is divided into four subblocks of $N_S/4$ bytes each. (Assume N_S is divisible by 4.) The first subblock, $S[0]$, is the *page request subblock*, and is used when requesting pages. The remaining three subblocks, $S[1]$, $S[2]$, and $S[3]$, are *fingerprint key* subblocks; any or all of these bits may be used in the secret key when computing the fingerprint of a page, as is done during during page reconciliation.

3.5.2 Block storage

A *block storage* stores the blocks and pages used for communication with a given partner c.

A client contains *two* block storages for each partner c: an *outgoing block storage* that contains the blocks and pages used for sending messages to c, and an *incoming* *block storage* that contains the blocks and pages used for receiving messages from *c*.

This division arises from the need to maintain separate outgoing and incoming one-time pads for each contact. If the contacts shared a single one-time pad, then a race condition could arise in which both parties attempted to send a message at the same time, using the same blocks from the pad. This would pose a security problem, as using the same one-time pad to encrypt two different messages reveals information about both messages.

Each block storage contains four collections of objects:

System block pool An unordered collection of system blocks and associated IDs.

Encryption block pool An unordered collection of encryption blocks and associated IDs.

Unreconciled page pool An unordered collection of pages that have not yet been reconciled, and their associated IDs and fingerprints.

Reconciled page list A FIFO queue of pages that have been reconciled. This is the only part of the block storage for which the ordering of the elements is important; both partners in a hyper-encryption scheme must have the same ordered list of reconciled pages in order to guarantee that they produce the same set of system and encryption blocks.

The unordered collections (pools) allow random access; retrieval and removal of an element can be performed by its ID. The only read operation the reconciled page list permits is the removal of the first page in the queue, and the only write operation it permits is the addition of a page to the end of the queue; these restrictions enforce the requirement that the list maintain its order.

3.5.3 Initiating communication

Before Alice and Bob can communicate, they must establish a shared secret. This shared secret is used to generate an initial set of system blocks.

Let α be the number of PSN pages required to make a page of the one-time pad, as defined in Section 3.5.6. Then the shared secret must be *at least* $N_S \cdot \alpha$ bytes long, in order to allow Alice and Bob to retrieve at least α pages from the PSNs. Because some PSNs may be unavailable, and some pages will be lost to reconciliation, secrets of up to 3α blocks ($3N_S\alpha$ bytes) are desirable. There is no upper bound on the size of the shared secret; longer secrets are more desirable because they translate into more system blocks, making it easier to obtain enough pages from the PSNs. However, longer secrets are less user-friendly, and may require more work to generate.

The shared secret may be generated by Alice and Bob in person or via some other private channel, using a true random source, such as one described in Section 2.1.

A more practical alternative

Given the potentially long length of the secret, it is probably more practical to temporarily relax our secrecy assumptions in exchange for increased usability. Alice and Bob can use a key agreement protocol, such as Diffie-Hellman key exchange, to establish the shared secret.

We assumed in Section 3.3.1 that the adversary has sufficient computational power to discover the key produced by the Diffie-Hellman exchange, but that it requires a few weeks to do so. Once Alice and Bob have used up the key to download pages, the adversary gains no benefit from discovering the key. He cannot discover any page that Alice and Bob both downloaded from an uncompromised PSN, because the page has been destroyed by the PSN. Thus, the pages are secure even if an adversary discovers the initial secret.

One more point to be made about use of Diffie-Hellman for this purpose is that it requires that Alice and Bob mutually authenticate somehow, in order to avoid a man-in-the-middle attack. They may work out some practical way to authenticate this initial contact, such as by a brief phone conversation.

3.5.4 Requesting and downloading pages

In order for Alice to send messages to Bob, Alice and Bob need common pages of random data. They obtain these pages by using system blocks to download pages from PSNs. Each system block is used to download one page and compute its fingerprint; these fingerprints are used during reconciliation (see Section 3.5.5) so that Alice and Bob can confirm that they have the same pages.

To download a page, user U chooses an arbitrary system block from her system block pool. Let s be the ID of the chosen system block, and $S[s]$ be the block itself. Let $P_U[s]$ denote the page retrieved by U using $S[s]$.

To retrieve a page, U takes the $N_S/4$-byte ($2N_S$-bit) page request subblock of $S[s]$ and splits it into two halves. The first N_S bits are used to select a PSN, N_j, in an agreed-upon manner. (Possible methods for doing so are described below.) The last N_S bits are used as the request key in requesting a page from N_j, as described in Section 3.4. The page so received becomes $P_U[s]$.

The purpose of the fingerprint is to enable Alice and Bob to tell if they have downloaded the same page. To this end, it must be infeasible for the adversary to find two different pages for which Alice would compute the same fingerprint. If the adversary could find such a pair of pages P_1 and P_2, he could configure a compromised PSN to serve P_1 in response to each request it has not seen before, and P_2 in response to any request it sees for a second time. This results in Alice getting P_1 and Bob getting P_2 (or vice versa). After reconciliation, they then incorrectly believe that they have the same page; the two will then end up with different OTPs and be unable to communicate.

The fingerprint is computed using the first p bits of the remaining subblocks of $S[s]$ as the secret key, for some agreed prime p. Because $S[s][1]$, $S[s][2]$, and $S[s][3]$ are

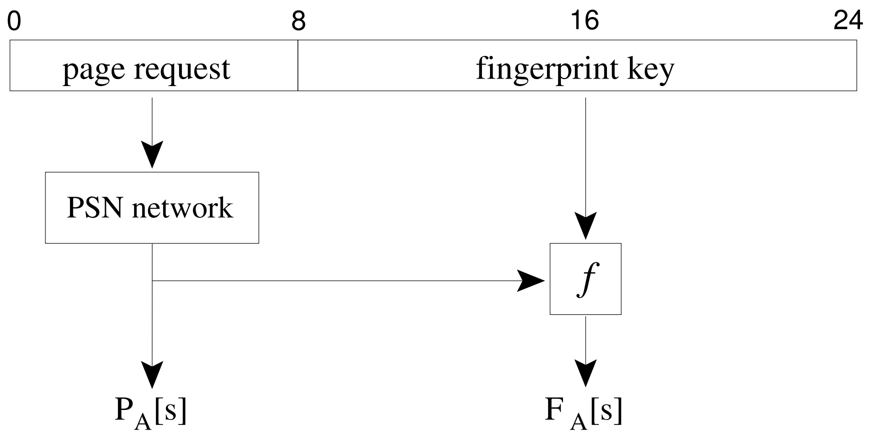


Figure 3-1: The process by which the system block with ID s is used to download a new page. The first system subblock (the *page request subblock*) is used to select a PSN and download a page of random data; this page becomes $P_A[s]$. The remaining subblocks are used to compute the fingerprint of the new page (represented by the block f); this fingerprint is saved as $F_A[s]$.

known only to Alice and Bob, an adversary cannot create two different pages with identical fingerprints with probability greater than $8N_P \cdot 2^{-p}$, where N_P is, again, the number of bytes in a page (and hence, $8N_P$ is the number of bits). Because the adversary cannot find two pages with the same fingerprint, Alice and Bob can be assured with high probability that if their page fingerprints match, then they obtained the same page from the PSN.

Once the page is downloaded and the fingerprint is computed, the system block $S[s]$ is discarded. If the page cannot be downloaded because N_j is not responding to requests or otherwise unavailable, $S[s]$ is set aside to be retried later.

The page downloading process described here is illustrated in Figure 3-1.

Choosing a PSN

Obtaining a page from a PSN first requires choosing which PSN to query. As described above, the client uses a N_S-bit word, the *PSN selection key* to choose a PSN.

The strategy we use in our implementation requires that each client have access to a list of PSNs (described by IP address, domain name, or some other method). Each PSN is associated with a numerical key. The client then uses the PSN whose key is numerically closest to the PSN selection key.

An advantage of using this nearest-neighbor strategy is that if Alice and Bob have similar, but slightly different PSN lists, most PSN selection keys will yield the same PSN for both of them. For example, suppose Bob's PSN list is identical to Alice's, except that some PSN N with key k is present in Alice's list and missing from Bob's list. When Alice looks up a PSN with a key close to k, she will find N, while Bob finds some other PSN. But for all keys not numerically close to k, Alice and Bob will get the same PSN.

The primary disadvantage of this strategy is that it requires clients to obtain a list of PSNs from somewhere. One method for publishing the PSN list is for a trusted authority to maintain a central nameserver that keeps a private copy of the list. Then, instead of clients keeping their own copies of the list, they send a PSN selection key to the nameserver, which responds with the IP address or domain name of the corresponding PSN.

Another method would be for a trusted authority to periodically publish the PSN list. This trusted party could make the PSN list available at a well-known website, or alternately, authenticated by simultaneously publishing a copy of the list in the *New York Times* (or some other such public source).

A downside of both of these methods is that they are single points of failure, and as such are vulnerable to a denial of service attack; a motivated attacker can attempt to take down or otherwise make unusable the central server(s) responsible for publishing the PSN list. While this can be mitigated by distributing the PSN list via a peer-to-peer service such as BitTorrent, it remains a concern.

Another downside is that the PSN list needs to be authenticated so that users can be sure that it originates with the trusted authority. If the PSN list is not authenticated, an attacker who has compromised some number of PSNs can publish a PSN list containing only the compromised PSNs, subverting the limited access model assumption as described in Section 3.3.1.

The problem of how to distribute a list of PSNs in an information-theoretically secure way is one that bears further study. As stated in the adversarial model in Section 3.3.2, we will assume that there is some trusted service from which Alice and Bob can securely obtain a list of PSNs.

3.5.5 Page reconciliation

Alice and Bob share the same set of system blocks. However, because they retrieve pages from PSNs asynchronously, it is possible that they receive different pages. That is, $P_A[s]$ and $P_B[s]$ are not necessarily equal for all s; it may even be the case that for a particular s, one or both of $P_A[s]$ and $P_B[s]$ does not exist because the corresponding PSN was not available when either Alice or Bob tried to contact it, or because one of the two didn't attempt to download it yet.

The *page reconciliation protocol* allows Alice and Bob to find and agree upon a common set of pages by publicly comparing the fingerprints of the pages they have downloaded. This protocol is reminiscent of the public discussion protocols employed in *BB84* [7] and Maurer's key agreement protocol in the "noisy channel" model [8].

Each stage of the reconciliation process consists of a single message from Alice to Bob or vice versa. Each message is sent unencrypted, but authenticated using the scheme described in Section 3.5.8 in order to protect against forgeries. The three stages are

1. **initiation**—Alice sends Bob a message containing the fingerprints of the pages she has downloaded;
2. **response**—Bob sends Alice a message telling Alice which of her page fingerprints match his own pages; and
3. **conclusion**—Alice reads Bob's response message and updates her own pages accordingly.

Initiation

To initiate the reconciliation, Alice produces a list, $[s_0, s_1, \ldots, s_{k-1}]$, of the IDs of each unreconciled page that she has downloaded. For each of these pages, she has already computed the fingerprint $F_A[s_i]$ of the page, incorporating part of the system block used to download that page (see Section 3.5.4). Alice sends the list of tuples $[(s_0, F_A[s_0]), (s_1, F_A[s_1]), \ldots]$ to Bob. This list is the *reconciliation initiation message.*

Response

To respond to a reconciliation initiation message, Bob takes Alice's list of page IDs $s_0, s_1, \ldots$, computes each fingerprint $F_B[s_k]$ as described above, then compares them to the fingerprints sent by Alice. There are three possible outcomes for each s:

- *success* if $F_A[s] = F_B[s]$;
- *failure* if $F_A[s] \neq F_B[s]$; or
- *no result* if he has not yet downloaded the page $P_B[s]$.

In the event of a success, Bob marks $P_B[s]$ as a reconciled page. In the event of a failure, Bob discards $P_B[s]$. Finally, Bob sends the list of results to Alice; this list is the *reconciliation response message.*

Conclusion

Alice concludes the reconciliation when she receives Bob's reply. For each s for which Bob indicated *success*, she marks $P_A[s]$ as having been reconciled. For each s indicated as a *failure*, she discards $P_A[s]$. For each s that Bob has declared as *no result*, she takes no action; $P_A[s]$ remains an unreconciled page, and may be included in a future reconciliation attempt.

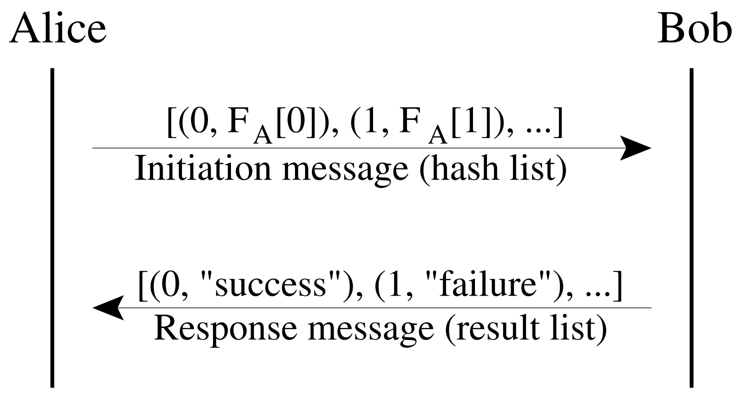

Alice
Bob
[(0, F_A[0]), (1, F_A[1]), ...]
Initiation message (hash list)
[(0, "success"), (1, "failure"), ...]
Response message (result list)

Figure 3-2: The page reconciliation protocol. Alice sends Bob a list of page fingerprints, and Bob replies to indicate which fingerprints matched his pages.

Dropped messages

Reconciliation messages may be transmitted over an unreliable medium where messages may be dropped. Messages may also be rejected if their MAC does not verify, either due to innocent corruption in transit or due to malicious modification. Consider the following scenario:

1. Alice sends a reconciliation initiation message to Bob.
2. Bob receives the message and processes it, marking Page 1 as having been reconciled, and discarding Page 2.
3. Bob sends a reply to Alice indicating that Page 1 was a *success* and Page 2 was a *failure*.
4. Bob's message is dropped by the e-mail server.

We now have a situation where Alice and Bob's reconciled page lists are out of sync, because Bob has processed a reconciliation message but Alice has not processed his response. If Alice and Bob remain out of sync, then when they combine their reconciled pages into OTP pages, they will wind up with different OTPs. At this point, they no longer have a shared secret pad, and the only recourse is to discard everything and start over from scratch with a new shared secret.

We can partially solve this by having Alice resend her initiation message if she doesn't receive Bob's reply within some timeout period. However, this causes another problem: when Bob receives Alice's re-sent message, he will process it differently! Using the example above, Bob's second response will now mark both Pages 1 and 2 as *no result*, because he no longer has any unreconciled pages with IDs 1 and 2.

To solve this, Bob needs to cache initiation messages received from Alice and the responses he sends back. When an initiation message arrives from Alice, if Bob finds it in his cache, he simply re-sends his original response.

Reordered messages

Care must also be taken in a medium where message order is not preserved, or in which the adversary has the ability to reorder messages (as we have assumed in Section 3.3.1).

Suppose Alice sends two initiation messages, M_1 and M_2. Bob receives and processes M_2 first, then M_1, and sends replies R_2 and R_1 to Alice. If Alice processes them in the original order (R_1, then R_2), then Alice and Bob will wind up with different reconciled page lists! In Bob's list, the page IDs enumerated in M_2 will come before those in M_1, and vice versa for Alice.

One solution is to assign unique IDs, in order, to each reconciliation initiation message, and require that reconciliation messages be processed in order. A simpler solution is to only allow one reconciliation to be pending at any given time. That is, Alice may not send another reconciliation initiation message until she has received and processed Bob's response. This ensures that no reordering of messages can occur.

We can modify the protocol slightly to avoid this problem altogether. The modified protocol, described in Section 5.1, is not yet implemented in our system.

3.5.6 Creating new encryption and system blocks

Alice and Bob now have a common ordered list of reconciled pages. Each uses these pages to produce encryption blocks and system blocks.

Define α to be the number of PSN pages that are combined to form a single N_P-byte page of the one-time pad (an "OTP page").

First, Alice removes the first α pages from the reconciled page list. She XORs them all together, obtaining a N_P-byte one-time pad, $OTP = P_A[s_1] \oplus P_A[s_2] \oplus \cdots \oplus P_A[s_\alpha]$. XORing the pages together ensures that even if the secrecy of some pages is compromised (for example, because of a faulty PSN), the resulting OTP remains secure; an attacker gains no information without knowing *all* of the α component pages. Under the limited access model assumption, an eavesdropper can subvert at most 1/5 of PSNs, so the probability of an attacker learning all α pages is at most $(1/5)^\alpha$. For a typical implementation, $\alpha = 30$, and $(1/5)^\alpha \approx 2^{-69}$. For those in possession of extreme amounts of paranoia, this probability may be made arbitrarily small by using a larger value of α.

Alice then takes the first $N_P/2$ bytes of this pad, and divides it up into blocks of N_S bytes each, thus producing $N_P/2N_S$ new system blocks. She appends these to her list of system blocks, assigning IDs to each block in some systematic way (for example, by assigning IDs in ascending order, beginning with the integer after the last ID assigned). She takes the remaining $N_P/2$ bytes of the pad and divides it into N_E-byte encryption blocks, and appends them to her list of encryption blocks, similarly assigning IDs to each. Bob does the same, independently of Alice.

Creating a single OTP page requires at least α PSN pages, and thus at least α system blocks. It may require more, because pages may be discarded through the page reconciliation process. Each OTP page creates $N_P/2N_S$ new system blocks. So long as these $N_P/2N_S$ new blocks can be used to obtain at least α new pages, to replace the α pages that were used, Alice and Bob can communicate in perpetuity.

In our implementation, $N_S = 32$, $N_P = 4096$, and $\alpha = 30$. This gives $N_P/2N_S =$

Process	SB	EB	UP	RP
Download a page	-1	0	$+1$	0
Successfully reconcile a page	0	0	-1	$+1$
Create new blocks	$+\frac{N_P/2}{N_S}$	$+\frac{N_P/2}{N_E}$	0	$-\alpha$
Encrypt an m-block message	0	$-m$	0	0

Table 3.1: A summary of how the various client operations affect Alice's outgoing block storage. The columns denote the four components of a block storage: the system block pool (SB), encryption block pool (EB), unreconciled page pool (UP), and reconciled page list (RP). Each number indicates a change in the relevant number of blocks or pages: a positive number means that new blocks/pages are added, while a negative number means that existing blocks/pages are expended and discarded.

64. Thus, if we assume that at least one in every two PSN pages survives reconciliation, then the 64 new system blocks will yield at least 32 new PSN pages. Because only $\alpha = 30$ are required, this is sufficient to generate another OTP page.

3.5.7 Sending messages

To send an encrypted message M to Bob, Alice breaks the message into m blocks of N_E bytes each. (Let us assume for simplicity that $|M|$ is a multiple of N_E bytes.) Let M_i be the ith block of M. Alice then produces a ciphertext C by encrypting each block with an encryption block; $C_k = M_k \oplus E[s_k]$ for each $k \in [0, m)$. The IDs of the encryption blocks $s_0, s_1, \ldots, s_{m-1}$ need not obey any particular ordering and need not be consecutive; the client may choose them arbitrarily from the available encryption blocks (but they must, of course, all be distinct).

Alice then sends to Bob the ciphertext C and the list of IDs $s_0, s_1, \ldots, s_{m-1}$. Bob decrypts the message by dividing C into N_E-byte blocks, and computing $M_k = C_k \oplus E[s_k]$.

Both Alice and Bob must destroy the encryption blocks used for this message once encryption/decryption is complete, to ensure that they are never used for another message.

3.5.8 Authentication

Attacks against unauthenticated messages

Hyper-encrypted messages and reconciliation messages need to be authenticated. An adversary who is capable of modifying messages in transmission can cause trouble if messages are not authenticated.

The simplest attack is to modify the ciphertext. For example, suppose Alice sends the encrypted message C to Bob. If an attacker knows the plaintext message M, then she can deduce the one-time pad $P = C \oplus M$. From there, she can change the message to an arbitrary message M' by replacing the ciphertext with $C' = C \oplus M \oplus M'$. Even if the attacker does not know the plaintext M, she can replace C with random data so that the decryption yields garbage.

A more subtle attack leaves the ciphertext unchanged, but modifies the list of encryption block IDs, causing Bob to decrypt the message incorrectly, using the wrong encryption blocks. This not only renders the message unreadable, but also wastes Bob's encryption blocks.

The adversary may also modify reconciliation messages, causing Alice and Bob to use different pages of data to form their OTP, and thus preventing their communication entirely.

Authenticating messages

Alice can authenticate any message to Bob by using part of the one-time pad as a secret key to compute a fingerprint of her message, using Rabin's fingerprinting scheme (Section 2.2).

To create a secret key for the fingerprint, she chooses k encryption blocks, concatenates them together, and discards the last d bits, choosing the smallest d such that $p = k \cdot N_E - d$ is prime. This process yields a p-bit key K. Let $s_1, s_2, \ldots, s_k$ be the IDs of the k encryption blocks used.

To send an authenticated, but *unencrypted*, message to Bob, Alice computes the fingerprint F_K of M, then sends Bob the tuple $(M, F_K, s_1, s_2, \ldots, s_k)$. To verify the fingerprint, Bob reconstructs the key K by referencing his own encryption blocks $E[s_1], \ldots, E[s_k]$, computes the fingerprint, and compares it to the F_K received from Alice.

If the fingerprint matches, he proceeds with the processing of the message. He also discards the blocks used to construct K.

If the fingerprint *does not* match, he discards the message, and *retains* the blocks used to construct K. An incoming message with an invalid fingerprint must leave the state of Bob's block storage unchanged, so that an adversary cannot affect Bob's state by sending bogus messages.

Authenticating a hyper-encrypted message requires an extra step, because both the message and the accompanying encryption block ID list must be authenticated. To authenticate this message, Alice simply combines the message and ID list by expressing each ID in the list as a 4-byte integer, padding with zeros if necessary, and appending the list to the ciphertext. Then, she proceeds as above.

Chapter 4

Implementation

This chapter describes our reference implementation of hyper-encryption. Our implementation is written in Java. The source code makes use of language features and standard libraries introduced with Java 6.0 (the current version at the time of writing), so Java 6.0 or later is required.

The complete source code is available at

`http://people.csail.mit.edu/juang/hyper-encryption/`

4.1 Overview and requirements

Our implementation includes both PSN and client software. The PSN is relatively simple—its only job is to respond to page requests—so the client comprises the bulk of the implementation.

The client software is designed to allow the user to manage hyper-encryption communications with many different contacts (partners) at once. The primary goal of the client software is to enable a user to easily send messages to and receive messages from multiple partners.

Among the desired features of the implementation at design time were:

- Support for multiple contacts: Software has the ability to separately manage several different one-time pads

- Persistence: State is stored to disk so that the user can shut down the software and restart it later without having to perform additional setup
- Mobility: Stored state is easily accessible so that the user can switch computers temporarily or permanently

Our implementation provides both persistence and mobility using the Berkeley Database Java Edition[1] (BDB JE), provided for free by Oracle. System blocks, encryption blocks, and pages of data are written synchronously to a database so that the current state of the one-time pad for each contact is always reflected on disk. The database is contained within a single directory on disk, so it can be easily moved between computers and run anywhere.

The portability of our implementation and BDB JE allows our software to be run on any computer with a recent version of Java. In fact, it is conceivable for users to place the hyper-encryption software and database on removable media, enabling them to use hyper-encryption anywhere.

4.2 Page Server Node

The PSN is a server that listens for inbound connections. A client sends a request to a PSN by opening a TCP socket to the PSN on port 48369, or some other agreed-upon port, and writing a single 32-bit integer, the page request key, to the socket.

The PSN reads in the request key from the socket and retrieves the appropriate page as a byte array. It then sends the page to the requester by writing these N_P bytes back to the TCP socket and closing the connection.

The low-level nature of the PSN request protocol allows for easy interoperation between different clients. In addition to our complete Java implementation, I have also written a simple PSN in Python (see complete source code in Appendix B) that

[1]`http://www.oracle.com/database/berkeley-db/je/`

interoperates seamlessly with the Java hyper-encryption client.

Our PSN's page database uses the twice-told pad (2TP) method described in Section 3.4.2. The queue and map are represented using standard Java collections (specifically, a LinkedList and HashMap).

4.3 Client

The client software has four major components:

- The HyperStorage class implements the block storage (Section 3.5.2) for a single contact, for a single direction of communication (outgoing or incoming). It is responsible for storing the system blocks, encryption blocks, unreconciled pages and associated hashes, and reconciled pages.

- The HyperCollector class implements the client processes: encryption and decryption algorithms, the page reconciliation protocols, and the processes for requesting and downloading pages from PSNs. The blocks and pages used for these processes are stored in HyperStorage objects. Each HyperCollector owns two HyperStorages for each contact: one for outgoing messages, and one for incoming messages.

- The HyperCommunicator class is responsible for sending hyper-encrypted messages and other protocol messages to other users, and retrieving incoming messages.

- The HyperGui class is a user interface, and also the controller of the system. It is responsible for presenting received messages to the user and allowing their decryption, allowing the user to compose and encrypt new messages, and visually presenting the state of the system, e.g., how many encryption blocks are available. The UI owns and controls a HyperCommunicator and a HyperCollector.

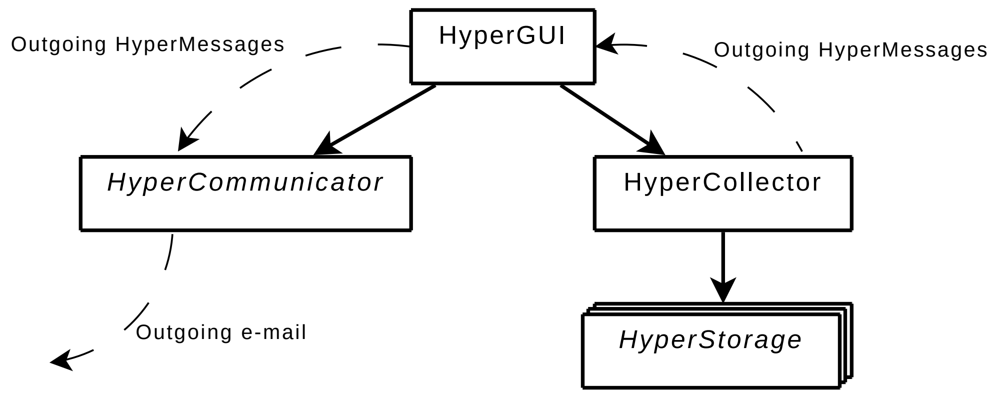


Figure 4-1: Simplified module dependency diagram for our Java hyper-encryption implementation. The HyperCollector uses HyperStorages (one per contact) to encrypt and authenticate outgoing messages, in the form of HyperMessages. These are passed by the UI to the HyperCommunicator, which translates them into e-mail messages and sends them out. Incoming e-mail messages follow the reverse process.

Of particular note, the communications functions and the encryption-related functions are completely separate; they interact only under the direction of the user interface (HyperGui), which owns both. The HyperCollector operates on abstract message objects called HyperMessages (described in Section 4.3.1). The HyperCommunicator is responsible for translating HyperMessages into some other format and sending them through some medium; in our implementation, each message is encoded into the body of an e-mail and sent from the user's e-mail account.

This relationship is illustrated in Figure 4-1.

4.3.1 HyperMessage objects

A HyperMessage represents the abstract notion of a message in hyper-encryption. This includes encrypted messages sent from one user to another, as well as reconciliation messages and responses. A HyperMessage has six required fields, a seventh optional field, and an eighth field applicable only to encrypted messages:

- *type*—one of *encrypted*, *unencrypted*, *reconciliation message* or *reconciliation response*

- *sender*
- *recipient*
- *date*
- *subject*
- *body*—this is the only part of the message that is actually encrypted
- *mac*—a HEMAC (see Section 4.3.2) authenticating the message (optional)
- *idlist*—a list of encryption block IDs (required only for *encrypted* messages)

All operations that work at the message level operate on HyperMessage objects. For example, the **decrypt** method in the collector takes an encrypted HyperMessage and returns a new HyperMessage containing the decrypted content.

The *mac* field is optional, but the default behavior of the client is to reject any incoming message that does not include a MAC. The most common case in which the *mac* field is omitted is when an incoming encrypted message is decrypted and stored locally. At this point, the MAC is no longer useful, or even meaningful, so it may be discarded.

HyperMessage objects are immutable. This design decision means that methods such as **decrypt** must construct new HyperMessages, rather than modify their arguments in place. This disadvantage is far outweighed by the benefit that immutability confers: it is easier to manage HyperMessages in a thread-safe way, and it also simplifies implementations of classes responsible for storing the HyperMessages to disk.

4.3.2 MAC

Hyper-Encryption MAC (HEMAC)

Section 3.5.8 described an authentication scheme based on Rabin fingerprints. Due to time constraints, our implementation does not currently support this fingerprint-based MAC; future versions will include this support.

Our current implementation uses a somewhat weaker construction related to

HMAC. It can be used with any cryptographic hash function h; let ℓ be the length, in encryption blocks ($8N_E$ bits), of the output of h. Our implementation uses SHA-256, which has $\ell = 4$ (because $N_E = 8$ bytes).

To compute the HEMAC, we use 2ℓ encryption blocks. Denote these encryption blocks by $E[0], E[1], \ldots, E[2\ell-1]$. Half of the blocks are concatenated and used as the key to HMAC. The result is XORed against the other half of the blocks, to guarantee that no information is revealed about X. Call the result of the computation Y:

$$Y = E[0:\ell] \oplus HMAC(E[\ell:2\ell], X)$$

The HEMAC is the tuple consisting of Y and the list of the IDs of the encryption blocks that comprise E. The notation $E[i:j]$ denotes the concatenation of encryption blocks $E[i], E[i+1], \ldots, E[j-1]$ (not including j).

This MAC can be used similarly to the fingerprint-based scheme. When Bob receives a message from Alice, he checks the HEMAC. If it is valid, he decrypts (if necessary) and proceeds. If it is invalid, he discards the message.

The **HyperMAC** class

A HyperMAC object represents a HEMAC, as described above. It has two fields:

- *value*—the MAC value
- *blocks*—a list of encryption block IDs used to create the key for this HEMAC

Our implementation uses SHA-256 as the underlying hash function for HEMAC. Thus, computing a HEMAC for a message requires first choosing eight encryption blocks (512 bits of data, which is twice the length of SHA-256). The IDs of these eight blocks become the *blocks* list. The blocks are then concatenated together to form the key for the HEMAC, and the *value* is computed as described above.

4.3.3 HyperStorage

The HyperStorage interface represents the *block storage*, defined in Section 3.5.2. It specifies operations for adding and removing blocks and pages from the block storage. The DBHyperStorage class provides the additional functionality of synchronously writing all changes to disk, so that the state of the block storage may be retained across sessions.

Desired properties

The HyperStorage implementation needs to have several properties:

Persistence The state of the block storage must be persistent across sessions. That is, the collections of blocks and pages should be stored to disk so that their state is saved when the user closes the hyper-encryption software or shuts down her computer, and can be restored the next time the software is run.

Synchronous I/O All writes to the block storage must be reflected on disk immediately. For example, the **remEncBlock** operation is required to write the removal of the block to disk *before* returning the removed block.

Thread safety HyperStorage objects should be safely usable by multiple threads at once.

The reason for the synchronous I/O requirement should be apparent with an example. If writes were not synchronous, the following sequence of events could occur:

1. A user encrypts a message using some encryption block, e.
2. User e-mails the encrypted message to a friend.
3. The power fails after the message is sent, but before the database updates are written to disk.

The block e was used in a decryption and should be removed from the block storage. But because of the power failure, that removal is not reflected in the on-disk state.

As a result, e will be used again to encrypt a future message! This reuse of one-time pad blocks poses a serious security problem.

To solve the problem, we need to be more careful about how updates are written to disk. It would probably be sufficient to use an *externally synchronous* model, similar to that described by Nightingale [12] for a local file system. This would require ensuring all writes to the block storage are flushed to disk before the effect of the write becomes externally visible, for example by the sending of an encrypted message via e-mail. For the sake of simplicity, our implementation uses a fully synchronous model, flushing every write to disk as soon as it happens. This behavior is clearly correct, and although synchronous I/O is expensive, we have not noticed a significant performance penalty.

Methods and API

The system block pool is implemented using a map; the key associated with each system block is its ID. The IDs are assigned in order, beginning with 1; each successive block added is assigned the next integer in sequence. The HyperStorage provides five methods for querying manipulating the system block pool:

- **int addSysBlock(byte[ ][ ] block)**

  The system block *block*, expressed as 4 subblocks of $N_S/4$ bytes each, is added to the system block pool. The block is assigned the next ID in sequence (that is, an ID exactly one higher than the last assigned ID). The method returns this ID.

- **byte[ ][ ] getSysBlock(int id)**

  Returns the system block with the given ID.

- **List<Integer> getSysBlockList()**

  Returns a list containing the IDs of all the system blocks in the system block pool.

- **int numSysBlocks()**

  Returns the number of system blocks in the system block pool. Equal to the length of the list returned from **getSysBlockList()**.

- **byte[ ][ ] remSysBlock(int id)**

  Removes and returns the system block with the given ID.

The HyperStorage allows retrieval and removal of system blocks in an arbitrary order, but controls over how IDs are assigned when blocks are added.

The encryption block pool provides the same operations, with two differences. First, encryption blocks are one-dimensional byte arrays of N_E bytes each, instead of an array of subblocks. Second, the encryption block pool provides an additional operation:

- **List<byte[ ]> remEncBlockList(List<Integer> idlist)**

  Removes and returns *all* the encryption blocks with the given IDs. This is an all-or-nothing operation: if *all* of the specified block IDs exist, then they are *all* removed and returned. If *any* of the specified IDs is missing, no blocks are removed, and an exception is thrown to indicate the failure.

This operation is useful for decrypting received messages. Each message comes with a list of encryption block IDs, all of which are required to fully decrypt the message. The **remEncBlockList** method allows the decryption method to atomically determine if the decryption is possible, and if so, do it.

Without this capability, the decryption method would need to take one of two alternative approaches:

1. Check first if every needed block is available, and then actually remove the blocks from storage. In this case, another thread may change the state of the storage between those steps. While it is unlikely that one of the required blocks would be removed, it is better not to take the chance.

2. Don't check first, and plow ahead with the decryption, removing each encryption block as it is used. If a required block is not found, then don't decrypt that part of the message, and either halt or attempt to decrypt the remainder. This is undesirable because it may leave a message only partially decrypted, which is confusing for the user and complicated for the software to manage.

The unreconciled page pool is also similar to the system block pool, except that rather than automatically assigning an ID to each added page, it requires that the caller specify the ID. The caller should choose the same ID as the system block that was used to download the page.

The **addUPage** method takes three arguments: an integer *id*, a *page* as a byte array, and a *hash* as a byte array. It stores the page and the hash, using the *id* as the key. The page and hash can be retrieved using the **getUPage** and **getUHash** methods. Finally, the **remUPage** method removes *both* the page and the hash with the given ID.

The reconciled page list is a FIFO queue, so it uses a different API from the collections described above. Because the reconciled page list does not allow random access, its methods do not use IDs. The supported operations are:

- **addRPage**—adds the given page to the end of the queue
- **remRPage**—removes and returns the page from the front of the queue
- **numRPage**—returns the length of the queue

Despite the different API, for simplicity of implementation DBHyperStorage actually uses the same underlying data structure for the reconciled page list: a Java SortedMap backed by synchronous writes to a Berkeley DB. The order of the pages in the queue is defined internally by their keys in the map (which are not part of the exported API). The **addRPage** method adds a page and assigns it the next ID in sequence. The **remRPage** method removes the page with the smallest ID.

Implementation

We fulfill the requirements of persistence, synchronicity, and thread-safety by storing blocks and pages in a Berkeley Database. Our implementation uses the Oracle Berkeley DB Java Edition (BDB JE). The BDB JE provides an API for managing a database that is compatible with the Java Collections API. This makes implementing the block storage relatively easy, as we can use a SortedStoredMap from the BDB JE to implement the block and page collections of the block storage. Our implementation then uses it as a standard SortedMap, and the BDB JE takes care of the database manipulations needed to actually store the blocks and pages in a thread-safe manner.

Our class PersistentMap serves as the interface between hyper-encryption client code and the abstraction layer of the underlying database. Our current implementation uses a StoredSortedMap from the BDB JE and exposes only the needed API to the client code. It enforces the synchronous I/O requirement by calling the **sync** method of the database environment every time a key-value pair is written to or removed from the map. Because it is inefficient to determine the size of a Berkeley DB stored on disk, PersistentMap reads the on-disk size only once, and then keeps track of the size in memory, updating it with every write operation. The PersistentMap utilizes appropriate locks to ensure that the size is updated in a thread-safe way.

4.3.4 HyperCollector

The HyperCollector is responsible for encrypting and decrypting messages, for processing incoming reconciliation messages/responses and creating outgoing ones, and for using system blocks to retrieve pages of data from the PSN network.

The HyperCollector owns two HyperStorages for each of the user's partners—one for outgoing communication, and one for incoming communication. These HyperStorages are stored in a pair of Maps, one for outgoing block storages, and one containing incoming block storages. Both maps hold key-value pairs where the key is the partner, and the value is the HyperStorage associated with that partner.

Methods and API

The HyperCollector class specifies seven methods corresponding to hyper-encryption protocols or processes. Two of the methods handle encryption and decryption of user-provided messages as described in Section 3.5.7:

- **HyperMessage encrypt(HyperMessage *m*)**

  Returns an encrypted version of *m*. The *type* of *m* must be "*unencrypted*". The body of *m* is encrypted, and its MAC computed, using the encryption blocks associated with *m.recipient*. The subject is *not* encrypted.

  The returned HyperMessage has the same *subject*, *sender*, *recipient*, and *date* as *m*. Its *type* is *encrypted*. Its *body* is the encrypted body of *m*, and its *idlist* is the list of encryption block IDs used to encrypt the body. The *mac* field contains the HEMAC (see Section 3.5.8) of the message, as computed by this method.

  This method affects the outgoing block storage for contact *c*. The encryption blocks used to encrypt *m* are removed from the pool of encryption blocks and discarded.

- **HyperMessage decrypt(HyperMessage *m*)**

  Returns a decrypted version of *m*. The *type* of *m* must be "*encrypted*". The body of *m* is decrypted using the encryption blocks listed in *m.idlist*.

  This method **does not** check *m.mac* for validity. It is the responsibility of the caller to verify the MAC and ensure the message is authentic, for example by using the **verifyHEMAC** method.

  The returned HyperMessage has the same *subject*, *sender*, *recipient*, and *date* as *m*. Its *type* is *unencrypted*. Its *body* is the decrypted body of *m*.

  This method affects the incoming block storage for contact *c*. If every required encryption block is available, then all the encryption blocks used in the decryption process are removed from storage and discarded. If any block is not available (that is, not in storage), then *no* blocks are removed; the decryption fails and this method throws an exception to indicate the failure.

Three methods implement the three stages of the reconciliation protocol as described in Section 3.5.5:

- **HyperMessage sendRec(Contact *c*, MessageParser *mp*)**

  Returns a HyperMessage of type *reconciliation message.* This is the first step in the page reconciliation process.

  The *body* of the message contains an ID-hash pair for each unreconciled page in the outgoing block storage for partner *c*. (Because the *body* of a HyperMessage must be a string, the provided MessageParser object, *mp*, performs the conversion between the list of ID-hash pairs and a textual representation.) The *subject* of the message is chosen arbitrarily, and is ignored by the methods that process reconciliation messages.

  This method computes the MAC of the message it produces, and places the computed MAC in the *mac* field of the returned HyperMessage.

  This message does not affect the block storages for contact *c*.

- **HyperMessage reconcile(HyperMessage *msg*, MessageParser *mp*)**

  Given a reconciliation initiation method as produced by **sendRec**, returns a reconciliation response HyperMessage containing the results of the reconciliation. This is the second step in the reconciliation process.

  The provided MessageParser is used to convert the body of *msg* to a list of IDs and hashes. It also serves to convert the results of the reconciliation, a list of page IDs and result codes, to a textual format.

  Similarly to **decrypt**, this method **does not** check the validity of *msg.mac*. The responsibility for verifying the authenticity of the message is left to the caller.

However, this method does compute the MAC of the message it produces, and places the computed MAC in the *mac* field of the returned HyperMessage.

This method affects the incoming block storage for contact *msg.sender*. Each page that is successfully reconciled is removed from the unreconciled page pool, and appended to the reconciled page list. Pages that fail reconciliation are removed from the unreconciled page pool.

- **void receiveRec(HyperMessage *msg*, MessageParser *mp*)**

  Process a reconciliation response message from contact *msg.sender*. This is the third and final step in the reconciliation process.

  Similarly to **decrypt**, this method **does not** check the validity of *msg.mac*. The responsibility for verifying the authenticity of the message is left to the caller.

  This method affects the outgoing block storage for contact *msg.sender*. Each page that was marked by *msg.sender* as successfully reconciled is removed from the unreconciled page pool, and appended to the reconciled page list. Pages that *msg.sender* marked as having failed reconciliation are removed from the unreconciled page pool.

One method pertains to verifying the MAC of a message:

- **boolean verifyHEMAC(byte[ ] *msgBytes*, List<Integer> *blocksUsed*, HyperMAC *expected*, Contact *c*)**

  Computes the MAC of the message *msgBytes* using the encryption blocks given in *expected.blocks* as the key, and compares the result to *expected.value*.

  If the *blocksUsed* list is not null, each integer in the list is appended to *msgBytes* as a big-endian four-byte array, and the MAC is computed for the resulting *msgBytes* array.

  If the MAC matches the expected value, this method returns true, and false otherwise. If the MAC cannot be computed, for example because one of the blocks specified in *expected.blocks* is not available, an exception is thrown (and the MAC is considered to have failed).

And finally, one method retrieves new pages from the PSNs:

- **void collectNext(Contact *c*, Direction *d*)**

  Gets a page from a PSN, using the data contained in a system block as described in Section 3.5.4.

  This method affects the block storage for contact *c* in the direction *d*. This process consumes one system block from the system block pool, and adds an unreconciled page with the same ID to the unreconciled page pool, along with the corresponding hash value.

  If unable to get a page from the PSN using a particular system block, for example because the PSN is not responding, this method tries each available system block until one succeeds (only the successful block is removed from the system block pool). If all blocks fail to produce pages, then this method throws an exception to indicate its failure.

4.3.5 HyperCommunicator

The HyperCommunicator performs tasks that require sending messages to or receiving messages from another hyper-encryption user. It converts between HyperMessage objects and formats suitable for transmission via another medium, such as e-mail or file transfer. It also carries out the actual act of transmitting and receiving said messages.

Our implementation includes a class named EmailHyperCommunicator, a Hyper-Communicator that encodes a HyperMessage into an e-mail message and vice versa. It sends these e-mail messages via SMTP, and retrieves incoming messages from other clients from an IMAP server. EmailHyperCommunicator uses the JavaMail API[2] to

[2] `http://java.sun.com/products/javamail/`

interact with IMAP and SMTP servers.

While our implementation focuses on transmitting hyper-encrypted messages via e-mail, other communication media are feasible. The HyperCommunicator interface allows for implementations that use some other medium. For example, a hypothetical HyperCommunicator might send a message by encoding a HyperMessage as an XML document and posting it to a world-readable website; it would receive messages by visiting each partner's website and downloading all messages addressed to the user.

Methods and API

HyperCommunicator specifies two methods, **send** and **receive**. EmailHyperCommunicator implements these methods:

- **void send(HyperMessage *m*)**

  This method encodes m into an e-mail message. The From: header is set to the user's e-mail address, and the To: header is set to the e-mail address of the recipient of m. The subject line of the e-mail is set to "Hyper-encrypted message" along with some identifying information; this subject line is ignored by the hyper-encryption client, but is provided so that a user viewing the (unintelligible) message in another e-mail client may recognize the message as being hyper-encrypted. This method sets the "X-Hyper-Encrypted:" header of the message; the hyper-encryption client recognizes incoming e-mail as being hyper-encrypted by the presence of this header.

  The contents of m are embedded into the body of the message in the format described in Appendix A. The message is then sent via an SMTP server, as configured by the user.

- **List<HyperMessage> receive()**

  This method retrieves all incoming e-mail from an IMAP server, as configured by the user. Messages without the "X-Hyper-Encrypted:" header are ignored, so that users may still send and receive regular, unencrypted e-mail without interference from the hyper-encryption client.

The bodies of the received messages are decoded according to the format specified in Appendix A, and the resulting list of HyperMessages is returned.

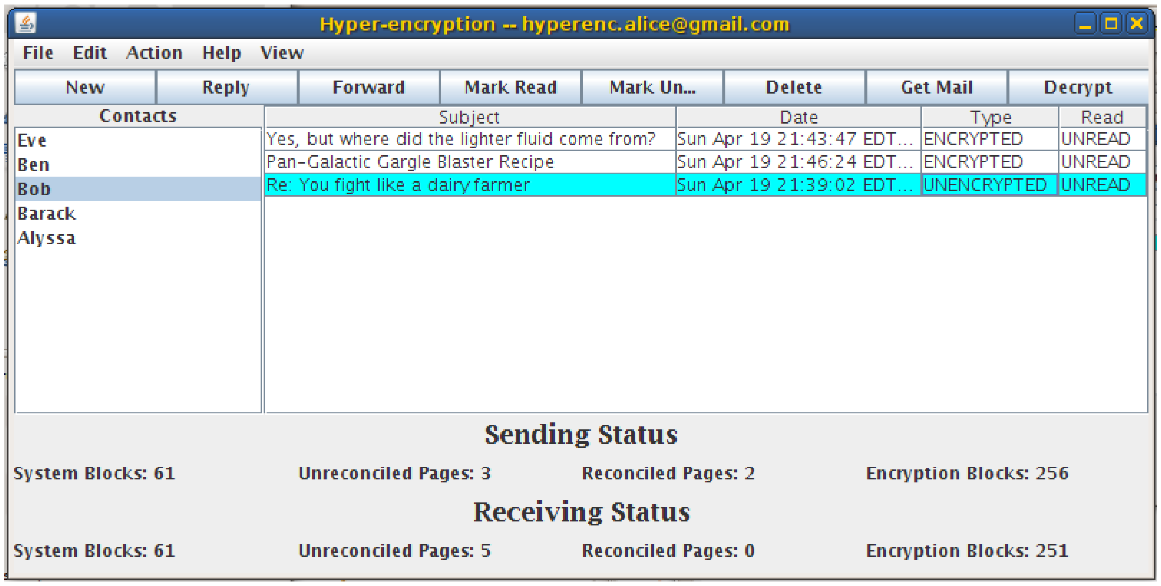
4.3.6 User interface

The HyperCollector and HyperCommunicator are independent objects. Neither directly interacts with the other, and, in fact, neither requires knowledge of the other. The user interface ties the two together and governs their interaction. The graphical UI included with our implementation is depicted in Figure 4-2. The UI has four primary jobs:

1. Manage the user's list of partners
2. Direct HyperCollector to retrieve pages, reconcile pages, etc.
3. Enable the user to compose messages, then encrypt and send them
4. Retrieve incoming messages, store them to disk, decrypt them, and display them to the user

The UI must maintain a list of the user's partners. Adding a new partner (also called a *contact* in our UI) requires creating new, empty HyperStorages for communication with that partner. The system block pool of the HyperStorage is initialized using the shared secret established by the partners.

Our GUI identifies partners by their e-mail address. It allows the user to specify a user-friendly display name for each partner, to be shown on screen in lieu of (or alongside) the e-mail address. When adding a partner, the user enters the shared secret as a hexadecimal string. This means that an n-byte secret requires $2n$ characters be entered. This makes typing in the secret unwieldy to the point of being impractical; the expected usage is that users will copy and paste the secret from a file.



Subject	Date	Type	Read
Yes, but where did the lighter fluid come from?	Sun Apr 19 21:43:47 EDT...	ENCRYPTED	UNREAD
Pan-Galactic Gargle Blaster Recipe	Sun Apr 19 21:46:24 EDT...	ENCRYPTED	UNREAD
Re: You fight like a dairy farmer	Sun Apr 19 21:39:02 EDT...	UNENCRYPTED	UNREAD

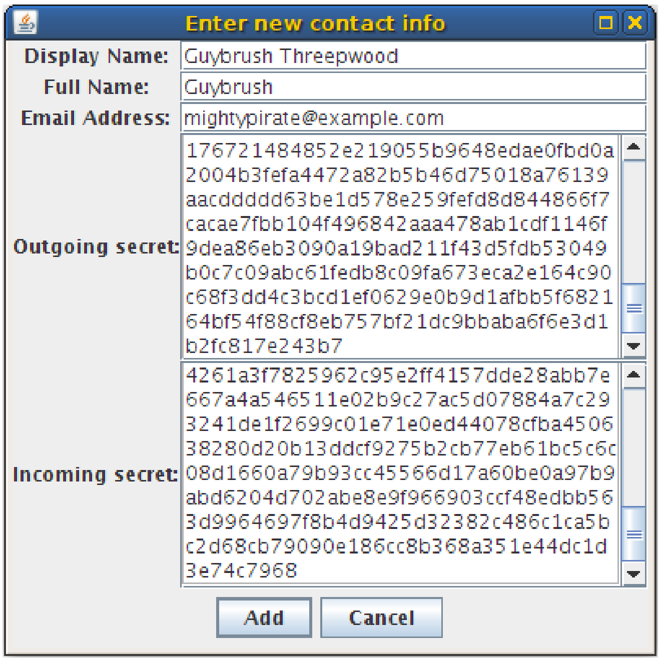
Sending Status

System Blocks: 61 Unreconciled Pages: 3 Reconciled Pages: 2 Encryption Blocks: 256

Receiving Status

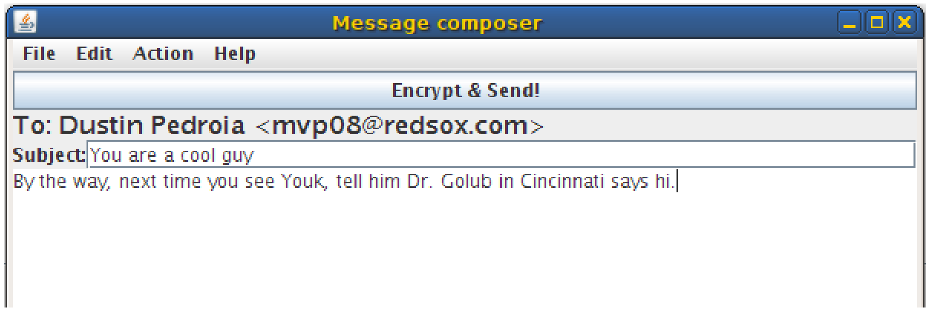
System Blocks: 61 Unreconciled Pages: 5 Reconciled Pages: 0 Encryption Blocks: 251

Figure 4-2: Screenshot of the GUI included with our hyper-encryption implementation. The left pane contains a list of all the user's partners. The right pane shows a list of all the messages received from the selected partner; in this picture, the selected partner is Bob.



Display Name:	Guybrush Threepwood
Full Name:	Guybrush
Email Address:	mightypirate@example.com
Outgoing secret:	176721484852e219055b9648edae0fbd0a 2004b3fefa4472a82b5b46d75018a76139 aacddddd63be1d578e259fefd8d844866f7 cacae7fbb104f496842aaa478ab1cdf1146f 9dea86eb3090a19bad211f43d5fdb53049 b0c7c09abc61fedb8c09fa673eca2e164c90 c68f3dd4c3bcd1ef0629e0b9d1afbb5f6821 64bf54f88cf8eb757bf21dc9bbaba6f6e3d1 b2fc817e243b7
Incoming secret:	4261a3f7825962c95e2ff4157dde28abb7e 667a4a546511e02b9c27ac5d07884a7c29 3241de1f2699c01e71e0ed44078cfba4506 38280d20b13ddcf9275b2cb77eb61bc5c6c 08d1660a79b93cc45566d17a60be0a97b9 abd6204d702abe8e9f966903ccf48edbb56 3d9964697f8b4d9425d32382c486c1ca5b c2d68cb79090e186cc8b368a351e44dc1d 3e74c7968

Figure 4-3: Screenshot of the Add Contact dialog in the GUI included with our hyper-encryption implementation. The shared secret is entered as a hex string.



To: Dustin Pedroia <mvp08@redsox.com>

Subject: You are a cool guy

By the way, next time you see Youk, tell him Dr. Golub in Cincinnati says hi.

Figure 4-4: Screenshot of the (minimal) e-mail composer in the GUI included with our hyper-encryption implementation.

Encryption and decryption of messages, as well as page retrieval and reconciliation, are delegated to the HyperCollector. The UI passes a HyperMessage or message body to the appropriate HyperCollector method (listed in Section 4.3.4), and displays the returned result to the user. This includes verifying the MAC of each message (using the **verifyHEMAC** method) and warning the user if any MAC fails.

Storing received messages to disk so that they persist across sessions is handled by the MessageStorage class. The GUI creates one MessageStorage for each of the user's partners. The MessageStorage works similarly to HyperStorage; it uses a Berkeley DB (via our PersistentMap class) to serialize and store GuiMessages to disk. A GuiMessage is simply a wrapper around a HyperMessage so that extra data specific to the user interface, such as user-assigned flags, and whether or not the message is unread, can be stored along with the HyperMessage. GuiMessages are immutable. This simplifies the code required to store them to disk, at the expense of incurring the cost of making a copy of the object every time the "unread" flag is toggled or other metadata is changed. Fortunately, this cost is relatively small, and does not seem to pose any significant performance penalty.

The client operations implemented by the HyperCollector, such as page retrieval, reconciliation, and producing blocks from pages, can be manually controlled by the user if so desired. However, because manually managing system blocks and pages may be daunting for the average user, and is definitely tedious for all users, the UI also offers the option of automatically invoking these processes whenever the number of blocks or pages available falls below preset thresholds.

4.4 Portability

Our hyper-encryption client and PSN are written entirely in Java. Even the Berkeley DB employed for saving data to disk is implemented in Java. As a result, hyper-encryption can be used on any machine with an up-to-date version of Java.

The Berkeley DB also provides portability in another sense. Because the entire database is contained within a single directory in the file system, it can easily be moved between computers. This allows a paranoid user, who believes that her home or office internet connection has been tapped, to take a copy of the hyper-encryption software and of her database somewhere else, and to download her pages or send/receive her messages there. She need not even use her own computer; she could keep the hyper-encryption software and data on a portable medium, such as a USB memory stick, and, as mentioned above, run it from any computer with a recent copy of Java.

4.5 Testing

Our implementation source code contains a suite of JUnit tests, with unit tests for major system components and utility classes.

We performed system-scale testing by creating a small hyper-encryption setup with three PSNs running on virtual machines (VMs), and a pair of clients exchanging messages through Gmail. Each of the VMs produces pages in a pseudorandom way, due to the lack of availability of a true random number generator; a PSN in a production system would, of course, need a truly random method of generating numbers, as discussed in Section 2.1.

We observed that running the PSN did not impose a prohibitive load on the ma-

Hostname	OS	RAM
`tourian.xvm.mit.edu`	Ubuntu 8.10	192 MB
`maridia.xvm.mit.edu`	Ubuntu 8.10	96 MB
`norfair.xvm.mit.edu`	Windows XP SP2	128 MB

Table 4.1: Specs of the three PSNs in our testing network. The three virtual machines all have somewhat limited amounts of power and resources, and so are reasonable approximations of older hardware.

chines. We used a script to flood the PSN with requests for random pages. On Maridia, the VM with the smallest amount of available RAM, the PSN process consumed 30% of physical memory (approximately 30 MB) in the steady state. The computational load on the machine was negligible.

On the same machine under the same conditions, our Python PSN implementation (see Appendix B for source) consumed 10.2 MB of virtual memory in the steady state, 7.6 MB of which was resident in physical memory (about 8% of the available physical memory).

In both cases, the relatively low memory usage combined with the observation that the machine remained responsive even under heavy load, leads us to conclude that the average user could run a PSN on her personal computer with little or no hindrance to normal usage.

The memory usage could be reduced if necessary by imposing a lower limit on the maximum number of pages the PSN may keep in memory, as discussed in 3.4.2. Our implementation's default settings permit up to 1000 pages to be resident in memory, for a total of 4 MB of page data (using our default setting of $N_P = 4096$ bytes). The remaining memory usage of the PSN process can be attributed to Java (or Python) overhead.

Chapter 5

Further Improvements and Future Work

There are several aspects of the hyper-encryption system that would benefit from further research. Additionally, parts of our implementation present opportunities for further development.

5.1 Improved reconciliation protocol

Section 3.5.5 discussed the potential problems that can arise if reconciliation messages are lost, or processed out of order. A modification to the reconciliation protocol can eliminate these problems. The original protocol relies on caching and resending of messages to ensure that Alice and Bob maintain a common ordered list of reconciled pages. The revised protocol proposed below eliminates the need for an ordered list, and instead gives Alice complete direction over how to combine reconciled pages.

In the modified protocol, Alice is responsible for *all* the bookkeeping; she sends Bob hash lists, and he replies, indicating which of the specified pages he has downloaded. Bob need not do anything else; he makes no distinction between pages that have been reconciled and pages that have not.

Alice remembers which of her pages are reconciled—that is, which pages Bob has, according to his reconciliation responses. When sending a message, Alice combines α of her reconciled pages and encrypts in the usual way. For each OTP page used in the encrypted message, she sends Bob the IDs of the PSN pages she used to create that OTP page; to decrypt, Bob combines the indicated reconciled pages, and then decrypts in the usual manner.

This improved protocol is not included in our implementation due to time limitations. Future versions of the hyper-encryption software will feature this improved method for reconciliation.

5.2 Method for selecting a PSN

As discussed in Section 3.5.4, the protocol for obtaining a page from the PSN network requires a method by which clients choose which PSN to use. That section suggested a method for doing so by using a list of PSNs with associated numerical keys. It also proposed several methods distributing such a list of PSNs in a secure manner, or for storing the list centrally and allowing clients to query it. Unfortunately, each of these methods is lacking either because it is not information-theoretically secure, is vulnerable to denial-of-service attacks, or both.

Securely distributing a list of PSNs or operating a nameserver that maps integers to PSNs may be feasible when hyper-encryption is used within an organization, such as a corporate environment, where authentication of the list is not an issue. But for general use across the broader internet, secure distribution of a PSN list poses a more difficult problem.

Hyper-encryption would benefit from an improved method for PSN selection, or from a more secure implementation of a method for distributing a PSN list.

5.3 Fine-tuning of parameters

One issue to be resolved in advance of a large-scale deployment is how to choose appropriate values for N_P, N_S, N_E, and α, as defined in Chapter 3. For now, we use the following values:

- $N_P = 4096$ bytes per page
- $N_S = 32$ bytes per system block (or 4 subblocks of 8 bytes each)
- $N_E = 8$ bytes per encryption block
- $\alpha = 30$ PSN pages combine to form one OTP page

These values were chosen somewhat arbitrarily; observation and analysis of a small-scale deployment would help determine if these values are appropriate in practice.

Our hyper-encryption client software will support automated retrieval and reconciliation of pages when the number of available blocks reaches certain thresholds. The default values for these thresholds will be chosen based on educated guesses. Refining these guesses based on typical usage patterns would help to improve the user experience.

Further, the hyper-encryption specification currently requires exactly $N_P/2N_S$ system blocks (which equals 64, in our current implementation) be created from each set of α reconciled pages. Only $\alpha = 30$ system blocks are necessary to retrieve these pages. This will probably lead to a surplus of system blocks: as more pages are downloaded to provide more encryption blocks for continued communication, system blocks will accumulate.

One solution to this is to devise and implement a method for dynamically adjusting what fraction of each page is made into new system blocks. When there is a system block surplus, fewer system blocks (and thus, more encryption blocks) are created from each page. Inversely, when the reserve of system blocks is running low, more system blocks can be created from each page, at the cost of creating fewer new encryption blocks.

5.4 Reducing the length of the initial shared secret

Currently, the initial shared secret must be at least $N_S\alpha$ bytes long, and must be longer in practice because some bytes will be wasted when pages are discarded via reconciliation. This is so long—at least $32 \cdot 30 = 960$ bytes in our implementation—as to preclude the possibility of typing in the shared secret; the most practical method is for partners to each have a copy of a file containing their shared secret, and then copy and paste the secret into the hyper-encryption software.

This poses a significant usability problem. One improvement would be to somehow shorten the minimum length of the initial shared secret key. One way to accomplish this would be to temporarily trade usability for security by reducing the value of α from 30 to 15, or even lower, for the first few rounds of hyper-encryption.

Another option would be to use a key exchange protocol, such as Diffie-Hellman key exchange, to automatically generate and enter the shared secret. This possibility is discussed in Section 3.5.3. It has the disadvantage of not being information-theoretically secure, but so long as the shared secret is completely used up before the adversary can discover the initial key, the retrieved pages remain secret. The boost to usability this would provide would greatly outweigh the limited risk of this scheme.

Chapter 6

Contributions

This document has presented hyper-encryption, a cryptosystem that is provably secure against unbounded computation within the limited access model. Hyper-encryption is information-theoretically secure, and furthermore, ciphertexts remain secret even if the initial secret key is eventually discovered by an adversary.

This thesis has described a Java implementation of the entire hyper-encryption system, showing that the system is workable in practice. In addition, it has discussed several potential vulnerabilities of the implemented system, and described strategies for minimizing their impact.

Finally, this thesis has described several improvements over the originally proposed hyper-encryption protocols, and proposed several others that have yet to be implemented.

Appendix A

E-mail message format

Our hyper-encryption client software encodes HyperMessages into e-mail messages using the format described in this appendix. This is a somewhat fragile format; future versions of the hyper-encryption software will support improved formats (XML-based, for example).

An e-mail message with this format represents a HyperMessage, and can be parsed according to this format to construct a HyperMessage. The EmailHyperCommunicator class implements this parsing in our implementation.

An e-mail message is identified to the software as being a hyper-encryption message by the presence of the `X-Hyper-Encrypted:` header. The address specified in the `From:` header corresponds to the *sender* field of the HyperMessage. The address specified in the `To:` header corresponds to the *recipient* field of the HyperMessage. If the e-mail Message has more than one recipient, all but the first are ignored.

The e-mail message body has five sections, which may appear in any order: *type*, *block list*, *MAC*, *subject*, and *content*. The sections may be separated by any number of newlines; leading and trailing whitespace in all sections is ignored.

The e-mail body contains a *subject* field containing the subject of the encoded HyperMessage. However, the subject line of *the e-mail itself* is ignored; we recommend it be set in a way that allows the user to easily identify the message as hyper-encrypted.

The *type* section starts with the header string "`*TYPE*`" on its own line. The next line contains a token indicating the type of the message. That token must be a string representation of one of the members of the enum HyperMessageType, e.g. "`ENCRYPTED`". This corresponds to the *type* field of the HyperMessage.

The *block list* section starts with the header string "`*PADSUSED*`" on its own line. The next one or more lines contain zero or more integers, each separated by any amount of whitespace (including newlines), identifying the IDs of blocks that were used to encrypt the message. If the block list is empty, this section must still be present, but contains only whitespace. These integers, in order, correspond to the *padsUsed* field of the HyperMessage.

The *MAC* section starts with the header string "`*MAC*`" on its own line. The next line contains a 256-bit HEMAC-SHA256, encoded as a 64-character hex string. The next line contains eight encryption block IDs, separated by spaces; these are the eight blocks used in computation of the HEMAC-SHA1, as described in Section 3.5.8. If the message does not contain a MAC, the MAC section must still be present, but contains only whitespace. This corresponds to the *mac* field of the HyperMessage.

The *subject* section starts with the header string "`*SUBJECT*`" on its own line. When parsing this section, leading and trailing whitespace is stripped, and newlines are replaced by spaces. This corresponds to the *subject* field of the HyperMessage.

The *content* section starts with the header string "`*CONTENT*`" on its own line. When parsing this section, leading and trailing whitespace (other than newlines) is stripped; newlines are preserved. This corresponds to the *content* field of the HyperMessage.

A.1 Sample e-mail message

This is an actual reconciliation response message produced by our implementation that illustrates the message format described above. Only the relevant e-mail headers are included. In this example, the *block list* section is present but empty because the message is not encrypted.

```
From: hyperenc.bob@gmail.com
To: hyperenc.alice@gmail.com
Subject: Hyper-encrypted message [type: SLAVE_REC]
X-Hyper-Encrypted: text

*TYPE*
SLAVE_REC

*PADSUSED*


*MAC*
539aa188d90ab38a1bcdc28b95c5de7a49e331c1d1a3b7e8abcf679c01f9dab2
25 24 27 26 29 28 31 30

*SUBJECT*
Slave reconciliation

*CONTENT*
80 +
81 +
82 +
83 +
84 +
```

Appendix B

Sample PSN implementation

This simple PSN is written in Python. It stores pages in a dictionary, keyed by their request key. If a request arrives with a request key that is not present in the dictionary, a new random page is generated on the fly and stored in the dictionary. If the request key *is* present in the dictionary, the associated page is removed and returned.

The code presented here uses Python's built-in `random` module, which uses a deterministic algorithm (specifically, the Mersenne Twister) to generate pseudo-random numbers. [13] As such, it is ***not suitable for production use***, and should be considered only as an example. A production-quality PSN would need to make use of a truly random source of numbers.

```
import socket
import random

DEFAULT_PSN_PORT = 48369

# Converts an string of 4 bytes to a 32-bit int, first byte
# being most significant
def bytes_to_int(b):
    b = map(ord, b)
    return (b[0] << 24) + (b[1] << 16) + (b[2] << 8) + b[3]
```

```
class PageDatabase:
    MAX_PAGES = 1000

    def __init__(self):
        # XXX this is a PRNG and is absolutely unacceptable
        # for our purposes
        self.rnd = random.Random()
        self.db = {}

    def getPage(self, key):
        page = self.db.pop(key, None)
        if page is None:
            # Generate 4096 random bytes and shove them all
            # together
            page = ''.join([chr(self.rnd.randint(0,255)) \
                            for i in range(4096)])
            self.db[key] = page
            self.flush()
        return page

    def flush(self):
        if len(self.db) > self.MAX_PAGES:
            self.db.popitem()  # discard an arbitrary page

def main():
    ss = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ss.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    ss.bind(('', DEFAULT_PSN_PORT))
    ss.listen(5)  # backlog of 5

    pd = PageDatabase()

    print "Listening on port %d..." % DEFAULT_PSN_PORT
    while True:
        (s, address) = ss.accept()
        s.settimeout(2.0)
        try:
            req_key_str = ""
            while len(req_key_str) < 4:
                req_key_str += s.recv(4 - len(req_key_str))
        except socket.timeout:
            print "Connection from", address, "timed out."
            s.close()
            continue
        req_key = bytes_to_int(req_key_str)
        print "Responding to request id %d." % req_key
        pg = pd.getPage(req_key)
        s.sendall(pg)

if __name__ == "__main__":
    main()
```

Bibliography

[1] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[2] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[3] Michael O. Rabin. Provably unbreakable hyper-encryption in the limited access model. *Theory and Practice in Information-Theoretic Security, 2005. IEEE Information Theory Workshop on*, pages 34–37, Oct. 2005.

[4] Ronen Shaltiel. Recent developments in explicit constructions of extractors. *J. Cryptol.*, 77:67–95, June 2002.

[5] M.O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Computer Science, DEAS, Harvard University, 1981.

[6] Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, 1949.

[7] C.H. Bennett and G. Brassard. Quantum cryptography: Public key distribution, and coin-tossing. In *Proc. 1984 IEEE International Conference on Computers, Systems, and Signal Processing*, pages 175–179, Dec. 1984.

[8] Ueli M. Maurer. Secret key agreement by public discussion from common information. *IEEE Transactions on Information Theory*, 39:733–742, 1993.

[9] Ueli M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. *J. Cryptol.*, 5(1):53–66, 1992.

[10] Christian Cachin and Ueli Maurer. Unconditional security against memory-bounded adversaries. In *In Advances in Cryptology CRYPTO 97, Lecture Notes in Computer Science*, pages 292–306. Springer-Verlag, 1997.

[11] Yonatan Aumann and Michael O. Rabin. Information theoretically secure communication in the limited storage space model. In *CRYPTO '99: Proceedings of*

the 19th Annual International Cryptology Conference on Advances in Cryptology, pages 65–79, London, UK, 1999. Springer-Verlag.

[12] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–1, Berkeley, CA, USA, 2006. USENIX Association.

[13] Python v2.6.2 documentation. `http://docs.python.org/library/random.html`, accessed May 2009.