

© Copyright by Adam James Slagell, 2003

A SIMPLE, PORTABLE AND EXPANDABLE CRYPTOGRAPHIC APPLICATION
PROGRAM INTERFACE

BY

ADAM JAMES SLAGELL

B.S., Northern Illinois University, 1999

M.S., Northern Illinois University, 2000

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

To the pursuit of knowledge.

Acknowledgments

I would like to thank Dr. Klara Nahrstedt for the invaluable direction and feedback she has given me on my first computer science paper and this thesis. She has provided the perfect balance of direction and freedom, allowing me to pursue my own ideas and supporting me the whole while. I would also like to thank my wife Amy who has been supportive of my many years of education and the time and resources it has cost. I would like to thank the other members of the MONET group, especially Chui-Sian Ong, who have provided feedback and help on numerous occasions.

Table of Contents

| | |
|---|-----|
| List of Tables | vi |
| List of Figures | vii |
| Chapter 1 Introduction | 1 |
| Chapter 2 Cryptography Survey | 5 |
| 2.1 Generic Block Ciphers | 6 |
| 2.1.1 DES | 6 |
| 2.1.2 In Search of AES | 7 |
| 2.1.3 Serpent | 8 |
| 2.1.4 MARS | 9 |
| 2.1.5 RC6 | 9 |
| 2.1.6 Twofish | 10 |
| 2.1.7 Rijndael | 10 |
| 2.1.8 Other Block Ciphers | 11 |
| 2.1.9 Block Cipher Modes | 12 |
| 2.2 Public Key Encryption Algorithms | 15 |
| 2.2.1 RSA | 15 |
| 2.2.2 ElGamal Cryptosystem | 16 |
| 2.2.3 Elliptic Curve Cryptography | 19 |
| 2.2.4 NTRU | 21 |
| 2.3 Specialized Video Encryption Algorithms | 23 |
| 2.3.1 VEA | 23 |
| 2.3.2 Permutation Algorithm | 23 |
| 2.3.3 I Frame Encryption | 24 |
| 2.3.4 SECMPPEG | 24 |
| 2.3.5 Zig-Zag Permutation Algorithm | 25 |
| Chapter 3 Related Work | 26 |
| 3.1 Cryptographic APIs | 26 |
| 3.2 Cryptographic Algorithm Implementations | 28 |

| | | |
|------------|--|----|
| Chapter 4 | Design and Implementation of Cryptographic Interface | 30 |
| 4.1 | Overview | 30 |
| 4.1.1 | Design Goals | 30 |
| 4.1.2 | API Structure | 31 |
| 4.2 | Class Details | 32 |
| 4.2.1 | Base Class: AKey | 32 |
| 4.2.2 | Symmetric Key Class: SKey Class | 34 |
| 4.2.3 | Public Key Class: PKey Class | 35 |
| 4.2.4 | An Example: RC6Key Class | 37 |
| 4.3 | Example API Use | 38 |
| Chapter 5 | Performance Results | 41 |
| 5.1 | Overview | 41 |
| 5.1.1 | Platforms | 41 |
| 5.1.2 | Block Cipher Tests | 41 |
| 5.1.3 | RSA Tests | 42 |
| 5.1.4 | ElGamal Tests | 42 |
| 5.2 | Block Cipher Comparisons | 43 |
| 5.2.1 | Key Generation | 43 |
| 5.2.2 | Encryption Performance | 44 |
| 5.2.3 | Decryption Performance | 45 |
| 5.2.4 | ECB vs CBC | 45 |
| 5.3 | RSA Performance | 47 |
| 5.4 | ElGamal Performance | 51 |
| Chapter 6 | Conclusion | 55 |
| Chapter 7 | Future Work | 57 |
| References | | 59 |

List of Tables

| | | |
|-----|--|----|
| 5.1 | Encryption - AES-128, Athlon XP 2000+ | 46 |
| 5.2 | Encryption - RC6-128, Athlon XP 2000+ | 47 |
| 5.3 | Encryption - Triple-DES, Athlon XP 2000+ | 47 |

List of Figures

| | | |
|-----|--|----|
| 5.1 | Here we plotted the key generation time in microseconds for different algorithms. AES and RC6 have both been tested with 128, 196 and 256 bit keys. DES always uses a 56 bit key, and Triple-DES was implemented with a 168 bit key. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test. | 43 |
| 5.2 | Here we plotted the encryption time in microseconds for different algorithms. AES and RC6 have both been tested with 128, 196 and 256 bit keys. DES always uses a 56 bit key, and Triple-DES was implemented with a 168 bit key. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test. | 44 |
| 5.3 | Here we plotted the decryption time in microseconds for different algorithms. AES and RC6 have both been tested with 128, 196 and 256 bit keys. DES always uses a 56 bit key, and Triple-DES was implemented with a 168 bit key. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test. | 46 |
| 5.4 | Here we plotted the key generation time in seconds for different RSA key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 25 iterations per test. | 48 |
| 5.5 | Here we plotted the encryption time in milliseconds for different RSA key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test. | 49 |
| 5.6 | Here we plotted the decryption time in milliseconds for different RSA key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test. | 50 |
| 5.7 | Here we plotted the key generation time in milliseconds for different ElGamal key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test. | 51 |
| 5.8 | Here we plotted the encryption time in milliseconds for different ElGamal key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test. | 52 |
| 5.9 | Here we plotted the decryption time in milliseconds for different ElGamal key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test. | 53 |

Chapter 1

Introduction

There are three main perspectives to information security. There is the mathematical perspective of the algorithms which is explored by algorithm designers and cryptanalysts. This may be the most discussed perspective, but it is rarely the weakness in any system. Usually it is not a brute force attack or vulnerabilities in the algorithms that cause security breaches, but rather how these algorithms are implemented in a larger system. The second perspective, whose importance is gaining recognition, is that of the user. Many users do not understand the semantics of cryptography nor care to understand. Most security problems come from this sort of user ignorance. For example, users often create poor passwords, or they do not understand what it really means when they sign and encrypt. For instance, if one encrypts first and then signs, it does nothing to indicate who wrote the original message. Someone else can strip off a signature, and add their own, never decrypting the message. But a user might assume the encrypt and sign operation is atomic or that the two operations are somehow commutative. The user aspect of security is difficult to address, as it really depends upon user education as well as making a good interface to the user. The third perspective is what our work addresses. This is the aspect of the application programmer. She usually is not a mathematician or cryptographer, but she is more familiar with cryptography than the average user. She desires an *Application Program Interface* (API) that is powerful and simple to use. A good cryptographic API allows a programmer to use cryptography without understanding any algorithm details.

Balancing features with simplicity is a difficulty in the development of any API. With each additional feature added, the learning curve to understand the API grows steeper. A minimalistic API will just provide core features. All intermediate or advanced features will need to be built from these basic features. This sort of API promotes code reuse but does little for the novice. A minimalistic cryptographic API would not be of use to a cryptography novice since she would likely need to write many intermediate features not provided. This would require a deeper understanding of cryptography. So while the API would not directly be a part of the large learning curve, we must consider the time spent to learn more about cryptography. An API can be simple but not minimalistic. Such an API will have all but the advanced features. As long as the API is well structured and documented, the novice will be able to learn it quickly. For most purposes, which will not need the advanced features, she can use the cryptographic API without learning much about cryptography. Lastly, there are the fully featured and very large APIs. These often become very complex and difficult to understand. At this point it does not matter whether one is a cryptographer or not, the difficulty is understanding the structure of the API. Just using it becomes cumbersome, and expanding it to include more cryptographic algorithms is exponentially more difficult.

All the APIs that we found were of the latter type, extremely large and full of extraneous features. A user will use few of these extra features, if any. Since cryptography is dynamic and the set of algorithms believed to be strong change over time, cryptographic APIs must be upgraded to include new algorithms. This is very difficult to do to a large API with a complex structure of dependencies between modules. Most of the APIs we came across suffer from this weakness due to their complexity. The Microsoft CryptoAPI suffers in this way more because it is closed source. Another important issue that is neglected by many cryptographic APIs is portability. Again, the Microsoft CryptoAPI suffers in this weakness.

Discovering short comings in many other cryptographic APIs, the following design goals were created in order of importance. The API must be simple for an application programmer to use. We want something at the level of simplicity that students could use in a course

on computer security for one or two projects. It must be simple to extend. Since the set of popular algorithms changes over time (due to advances in cryptanalysis mostly) this API must be easily extensible to include new algorithms. The API should also be platform independent and portable. The API should be general, only specifying primitives. This allows its use for many purposes beyond that of its original design. This also helps to meet the first goal of simplicity. Lastly, we want an API that can replace use of the Microsoft Cryptographic API with as little recoding as possible.

We created the API using a hierarchy of abstract key classes written in C++. The user of the API needs only to learn the structure of these classes all contained in one small header file. Expandability is enabled by the fact that features common to all algorithms are pushed up as high into the hierarchy as possible. This way a programmer need not rewrite code common to ciphers of a particular type. There is copious documentation within the code to see how it works, including templates to some common functions that will need to be written for new algorithms. Since there are already six encryption algorithms included, there are plenty of examples to look at for a programmer wishing to add a new algorithm. By writing the API entirely in standard C/C++, we have achieved portability. It builds either as a Windows DLL or a Linux .so. Porting it to other UNIX systems that use gcc should be trivial, requiring changes only to the makefile. Lastly, by making the interfaces to the methods similar to the Microsoft CryptoAPI, code using the MS API can be made to use ours with little modification. The main difference would be that one would no longer pass handles to keys but rather pointers to them.

This thesis is structured into the following sections. Section 2 is a survey of cryptography basics. Section 3 discusses related work. This includes the discussion of both other cryptographic APIs and specific algorithm implementations. Section 4 describes the API and the rationale behind design decisions. It also provides an example of its use. Section 5 presents some performance results and information learned from these results. This information influenced several changes in implementation that further increased performance. Section 6

discusses future work, and section 7 presents our conclusions.

Chapter 2

Cryptography Survey

There are two main types of cryptography: *symmetric/secret key*) and *asymmetric/public key*). Symmetric key cryptography is named so because the decryption operation is essentially the same as the encryption operation. One operation is not harder than the other and often just involves running the algorithm in “reverse”. In some symmetric key cryptosystems, just running the encryption algorithm again will decrypt the cipher-text. An example of such an algorithm would be simply **XOR**ing the data with a key. Also, common to all symmetric cryptosystems is the use of a single key which must be kept secret, hence the alternative term: secret-key cryptography.

Asymmetric cryptosystems have two keys, a private and a public key. Senders need only know the recipient’s public key. This key need not be secret, hence the term public key cryptography. The term asymmetric comes from the fact that anyone can encrypt rather easily, but without the secret key, decryption is extremely difficult. Asymmetric algorithms are based off of mathematical functions that have very difficult inverses to compute, without a “trap door”. The “trap door” can be exploited with knowledge of the secret key. Ideally, the inverse will be an NP-Hard problem whereas encryption will be in polynomial time.

Within these two classes there are many subclasses including, block ciphers, stream ciphers and special purpose encryption algorithms that are dependent upon characteristics of the plain-text.

2.1 Generic Block Ciphers

Generic block ciphers form a very important class of symmetric algorithms. These algorithms are independent of the type of data being encrypted. Block ciphers work independently upon each block, which is of a fixed size. Some algorithms allow a variable block size, but the block size is still made constant throughout the entire encryption session. When we discuss implementation details later, we will see that there are in fact different modes in which to operate block ciphers that can create dependencies between blocks.

Ciphers which do not break the encryption process into blocks are called stream ciphers. We chose to avoid stream ciphers in our cryptographic API because our original focus was on securing video streams. With stream ciphers, lost packets can cause resynchronization problems. Since the model for multicast video often allows for some packet loss, we decided it best to avoid stream ciphers. However, there are cases when they are appropriate, and being that the API has been generalized past use for video only, we would like to see a stream cipher class added in the future.

There are many block ciphers that have been developed and tested since the release of DES over 30 years ago. Still, many software packages only support DES, which is far too insecure, or triple-DES which is far too slow. There are many modern block ciphers which are both faster and far more secure than DES. Additionally, they often work with variable key sizes, allowing a user to trade speed for security. It is important for software developers to be made aware of some of these other choices and their corresponding advantages and disadvantages.

2.1.1 DES

In the early 1970s, the National Bureau of Standards (NBS) set forth to find a symmetric encryption algorithm to declare as a national standard. This algorithm was to be called DES (Data Encryption Standard). Eventually, it was decided to accept a variant of Lucifer,

developed at IBM, as the new standard. We say variant, because to the uneasiness of the public, the tables were modified by the NSA. Some felt that these modifications were made as some sort of trap door that allowed the NSA to defeat the algorithm [6].

This standard has persisted for over 30 years and become quite antiquated. To the shock of many, a company by the name of the Electronic Frontier Foundation made a serious blow to DES in 1998. They created a machine costing \$250,000 that could crack DES in only 56 hours [8]. Five years later we would expect that a machine capable of this could be made within the budget of many individuals or small companies. Also, with the gaining popularity of distributed computation software (such as the projects to find new Mersenne primes or analyze radio signals for SETI) the security of DES is even more questionable. In fact, there are often distributed computation projects to crack encryption algorithms. Millions of users dedicate otherwise idle CPU cycles to work on these projects for months at a time. Such projects make cracking DES ciphertext a trivial project.

To reinforce (or some would say patch up) DES, Triple-DES was created. Triple-DES, also denoted 3DES, is quite simple. Triple-DES generates three keys, and a message is encrypted with the first key, decrypted with the second key, and encrypted again with the third key. This has a profound effect by essentially tripling the key size and hence cubing the size of the keyspace. Triple-DES is the default option for much of the software employing cryptography now, even though a completely new encryption standard has replaced DES.

2.1.2 In Search of AES

Realizing that DES was no longer appropriate for many cryptographic applications, National Institute of Standards and Tables (NIST) set forth to find a new algorithm as the nation's Advanced Encryption Standard (AES). The search began in 1997. NIST set forth several criteria for an appropriate algorithm. The algorithm must be a symmetric block cipher capable of handling key sizes of 128, 196 and 256 bits. Secondly, the algorithm must be available worldwide on a royalty free basis. Simplicity was considered good since it would

aid in the understanding and cryptanalysis of the algorithm. Another requirement was that the algorithm would be suitable on many platforms and for many purposes. It was to be appropriate for 8-bit smart cards, ATM networks, HDTV, voice and satellite communications. In addition to working well on a variety of platforms, it was also decided that performance on the IA32 architecture was of particular importance. Successful algorithms were also to demonstrate resistance to both linear and differential cryptanalysis. In theory at least, it has been shown that DES is vulnerable to linear cryptanalysis, a method of attack unknown at the time of its design. Being vulnerable to differential or linear cryptanalysis may simply mean that patterns are observed, but it does not always mean that there are attacks to exploit these patterns.

In all, there were 21 submissions for the standard, 15 of which met NIST's basic criteria. These were Loki97, Rijndael, CAST-256, DEAL, FROG, DFC, Magenta, E2, CRYPTON, Hasty Pudding Cipher, MARS, RC6, SAFER+, Twofish and Serpent. Many of these were actually from abroad. In August of 1999, NIST announced the five finalists to be MARS, RC6, Twofish (all from the US), Rijndael (from Belgium), and Serpent (from U.K., Israel and Norway). It was from these five that we decided to focus upon, in searching for algorithms to implement. Since good performance on IA32 architectures, our test bed, is one of NIST's criteria for AES, we thought these finalists were a good starting place to look for candidates of our own. Also, since the AES candidates must be general and applicable to a variety of purposes, they are suitable for a general cryptographic API where we have no way to predict all of its potential uses. More information on the history and goals of the AES project can be found in [8].

2.1.3 Serpent

Serpent is similar to DES in its use of many XOR and shift operations. It uses S-boxes derived from the DES S-boxes. S-boxes are tables used to perform substitutions and are found in most block ciphers. Additionally, the key schedule is quite simple for Serpent.

However, Serpent's security over DES is mainly from its use of 32 rounds instead of 16 rounds as in DES [8]. One wonders why they did not simply increase the number of rounds in DES. Each round performs a substitution and permutation, so additional rounds usually complicate cryptanalysis. The downside is that having so many rounds makes Serpent the slowest of the finalists by a good measure [9]. This immediately eliminated it as a candidate for our needs.

2.1.4 MARS

MARS was developed at IBM and falls into the category of Feistel networks, the implication being that one does not need to implement two different algorithms for encryption and decryption. The key schedule that determines how the key is used in each round just needs to be adjusted. It also has 32 rounds, but of two types. The first and last 8 rounds employ **XORing**, addition and rotation. The central 16 rounds are more complex and utilize **XORing**, multiplication and data dependent rotations [8]. It is these more complex operations in the central rounds that make it significantly faster on CISC architectures than on RISC architectures [9]. One nice thing about MARS is that the key setup time and encryption speed are independent of the key size [8]. This ability to scale well does make MARS an attractive choice.

2.1.5 RC6

RC6 is another creation of Ronald Rivest at RSA Securities. It is a 20 round Feistel cipher and much simpler than MARS [8]. For this reason RC6 outperforms MARS most of the time. It also meets the goal of simplicity better. RC6 is similar to MARS in the use of many 32 bit multiplications and data dependent rotations. Because of its extensive use of such operations, it too will perform much better on CISC architectures such as the IA32. In fact, RC6 performs the best of all the finalists on Pentium II chips [9]. Like MARS, the key

scheduling is simple and the key setup and encryption speeds are independent of key size [8]. Additionally, optimizations in assembler can be made to RC6 that increase the performance over what is reported in [9]. These enhancements are presented in [10]. All in all, RC6 offers everything that MARS does and more. Since our test bed is on the IA32 architecture, we decided to implement RC6 as one of the symmetric algorithms.

2.1.6 Twofish

Twofish is another creation of the infamous Bruce Schneier of Counterpane Systems. It is basically a rework and improvement of his Blowfish algorithm, and he recommends Twofish over Blowfish in all cases. This may be from the fact that Blowfish has some weak keys, but we were unable to find any research directly comparing the two algorithms. Twofish differs from many other block ciphers in the use of dynamically varying S-Boxes. The algorithm is relatively complex, and that was one of NIST's main concerns with it [8]. Key size affects the key setup time, but it does not affect the encryption speed. On average Twofish underperforms Rijndael. It is not until 256 bit keys are used that Twofish begins outperforming Rijndael. Unlike MARS and RC6, it performs well on both CISC and RISC architectures [9]. There are few drawbacks to Twofish, other than its complexity, which makes it difficult to create solid statements about its resistance to certain cryptanalysis attacks. Only time will tell whether Twofish is really resistant to cryptanalysis.

2.1.7 Rijndael

The name is almost as interesting as the algorithm itself. It is a concatenation of parts of the names of both developers. Right away Rijndael sets itself apart as something different. It relies directly on a particular representation of the Galois field of order 256. This field consists of polynomials of degree less than eight with coefficients in Z_2 . Z_2 is the field which has only the elements **0** and **1**. Operations of multiplication and addition behave as they do

when considered as integers but with the result being taken modulo 2. Thus $1 + 1 = 0$ in Z_2 . An example of an element in the Galois Field of order 256 is $x^7 + x^5 + x^2 + 1$. Results of operations on polynomials are taken modulo an irreducible polynomial of degree eight. An irreducible polynomial is one that cannot be factored. Because we take the result of an arithmetic operation modulo a degree eight polynomial, the result is always of degree less than eight. Hence this field is closed under the operations of addition and multiplication. The branch of mathematics that Galois fields fall under is called abstract algebra, which is much more commonly used in public-key algorithms.

Having this algebraic structure, its operations are quite different than the other finalists [8]. Overall performance across many platforms seems to favor Rijndael. Surprisingly, it is still simple enough to work on very basic smart cards. Not even RC6 can do this because of memory constraints. The downside is that both key setup and encryption speed grow with key length. Encryption is about 20% slower on 196 bit keys and 40% slower with 256 bit keys. These numbers are comparing to the 128 bit speeds. However, it was mentioned earlier that this speed reduction seems to make little difference unless 256 bit keys are used. Then Twofish may be a better choice [9]. For these reasons and the fact that it is the new standard, we decided to implement Rijndael over Twofish. We would expect similar performance results from both. There is also a certain aesthetic appeal to an algorithm that uses interesting algebraic structures rather than simple XOR operations, rotations, additions and multiplications.

2.1.8 Other Block Ciphers

In addition to the AES candidates, other ciphers were considered. IDEA (International Data Encryption Algorithm) is a block cipher that was popularized by the commercial version of PGP. In 1996, when [6] was written, Schneier's opinion was that IDEA was the most secure block cipher available. With a 128 bit key, a software version of IDEA performs about twice as fast as 56 bit DES [6]. We would have examined IDEA more closely had it not been for

a patent against its commercial use. While this would not affect our work directly, we feel it best to focus on algorithms available for unrestricted public use.

CAST-128/256, named after its creators Carlisle Adams and Stafford Tavares, was popularized by Entrust and was one of the first alternatives to DES and 3DES in the SSH protocols. CAST-256 was one of the 15 first round AES candidates. It is demonstrated in [11] that CAST is resistant to both linear and differential cryptanalysis. One thing different from many of the other block ciphers is that the S-boxes are not fixed, but rather they are implementation dependent and the construction is quite complicated [6].

Blowfish, a Bruce Schneier creation, has been popularized mostly through its use as a password hash in OpenBSD. It was also one of the first DES and 3DES alternatives in SSH. This is not surprising since OpenSSH is developed by the same group as OpenBSD. Like MARS, it falls into the category of Feistel networks. It uses a 128 bit key size. One security concern with Blowfish is the discovery of some weak keys. Here a weak key means one in which two entries for a given S-box are identical. However, no exploit of the weak keys was known as of the writing of [6]. Conversing with Bruce Schneier, he said that in all cases Twofish is a superior algorithm and recommends its use over Blowfish. So there was no real reason to examine Blowfish further.

2.1.9 Block Cipher Modes

In addition to choosing the block ciphers to use, there is a choice of modes in which to operate them. All of the information about these modes can be found in [6]. The most straightforward way to implement a block cipher is in *Electronic CodeBook* (ECB) mode. In this mode every block is encrypted independently. It is fast, and blocks can be decrypted out of sequence. One has the choice of padding to an integral number of blocks or using ciphertext stealing. Ciphertext stealing is an alternative to padding that produces ciphertext of the exact same length as the plaintext, regardless of whether the plaintext is an integral number of blocks. Ciphertext stealing uses some extra **XOR** and swapping operations intermixed

with the encryption of the last full block and the partial block that follows. This has the side effect that a dependency is created between these last blocks. Ciphertext stealing can be used whenever there is more than one block, even if there is just one block and one byte. Our implementation of ECB uses cipher text stealing when there is more than one block of data. If there is strictly less than one block of data, padding is done. An alternative to ciphertext stealing would be to always have one extra byte at the end of the ciphertext to indicate how much padding was used on the last block. This would always allow the receiver to detect how long the original message is regardless of whether the ciphertext is less than one block size or not. However, that case is rare and impossible, as we will see, in *Cipher Block Chaining* (CBC) mode. Thus we chose cipher-text stealing which seems cleaner.

The other mode we implemented is CBC mode. Here, every block depends upon every previous block, thus complicating cryptanalysis. The problem is that blocks must be decrypted in order, which means the damage of one block is detrimental to all following blocks, and it is slightly slower. In CBC mode, the previous block of ciphertext is **XORed** with the current block of plaintext before encryption. This means that every message that begins with the same plaintext for n blocks has the same ciphertext for the first n blocks. The solution to this problem is the use of an *Initialization Vector* (IV) as the first block of ciphertext. This is just some pseudo-random block. It need not even be secret. As long as different IV's are used for different messages, ones that begin the same will have completely different ciphertext. I implemented CBC with IVs and ciphertext stealing. Because of the extra IV block, padding is never needed in this mode.

There are two other common modes in which to operate block ciphers: *Cipher-FeedBack* (CFB) and *Output-FeedBack* (OFB) modes. CFB mode acts more like a self-synchronizing stream cipher. Encryption can be done on any subblock size that is a divisor of the block size, even bit by bit. A smaller subblock size means more work, but the benefit is that the smaller subblocks contain the effect of errors in the ciphertext more locally. This mode requires an IV which must be different for different messages. OFB mode converts a simple block cipher into

a synchronous stream cipher. A synchronous stream cipher has a key-stream independent of the message stream. Both the sender and the receiver generate identical key streams. Loss of ciphertext data will cause the cipher to lose synchronization and decrypt everything incorrectly until the key-streams are resynchronized. This is actually a benefit since it will detect insertions and deletions in the message stream. Another benefit is that bit errors are not propagated at all. As important to security as key size is the *key-stream period*. A small period will greatly simplify cryptanalysis. The actual implementation details of CFB and OFB modes are similar. OFB also needs unique IVs, and it is slower than ECB or CBC. It should also be noted that OFB and CFB are meant for hardware implementations and make abundant use of shift registers. Done in hardware, the speed difference should not be that great from simple ECB. Another caveat is that inefficient generation of IVs will have a huge impact on the speed of an algorithm if the messages are small. This is a lesson learned first hand. Luckily the randomness of IVs is not nearly as important as the randomness of the keys. Thus randomness and speed can be traded off with IV generation.

There are two major ways to implement triple encryption for block ciphers. This choice was necessary since we implemented Triple-DES code from DES code primitives. *Inner-CBC* mode uses three IVs and encrypts the whole message three times. *Outer-CBC* mode uses one IV and encrypts each block three times and then proceeds. The benefit of inner-CBC is that it can easily be parallelized in hardware. However, it has been found that inner-CBC mode is only slightly more secure than than single encryption due to its vulnerability to differential cryptanalysis. Another choice with triple encryption is whether to *Encrypt-Decrypt-Encrypt* (EDE) or *Decrypt-Encrypt-Decrypt* (DED). It is arbitrary which mode is chosen, but EDE seems more common. We implemented triple DES in both CBC and ECB modes or more specifically outer-ECB-EDE and outer-CBC-EDE modes.

2.2 Public Key Encryption Algorithms

Common to most key distribution algorithms is the use of public key encryption, if for no other reason than authentication. Public key or asymmetric cryptography is significantly slower than symmetric cryptography. While there is significantly less data being encrypted using asymmetric algorithms, it can still be enough to add significant load to a server that is both acting as a key server and a sender of the data stream. Therefore, it is important to be aware of the options for public key algorithms.

2.2.1 RSA

RSA is one of the first (and by far the most prevalent) public key algorithms. The name stands for the three creators: Rivest, Shamir and Adleman. All the information discussed here can be found in more detail in [12].

Part of the appeal of RSA is its simplicity. It is based on the age old problem of factoring large integers into powers of primes. It is conjectured that RSA is as difficult to break as it is to factor large integers. In fact, there exists a very fast probabilistic algorithm that given a solution to the RSA problem (the public and private key), it can factor the modulus. Since factoring has been analyzed for centuries and is still deemed a difficult problem, RSA is considered to be secure. Of course there is always the possibility that someone will someday discover a significantly improved method of factoring. A similarly surprising result was proved this past summer. It was shown that primality testing can be done in deterministic polynomial time. This was long thought to be a far more difficult problem than it really is.

To generate a public/private key pair, a user, say Alice, first generates two very large primes. The use of the word “very” is subjective and its meaning significantly affects the security of the algorithm. Typically, RSA keys are at least 1024 bits long. Let p and q be the two large primes and $n = pq$. Next Alice computes the Euler-Phi function on n . Since n is a product of two primes, $\phi(n) = (p - 1)(q - 1)$. Next, she chooses an integer $1 \leq e \leq \phi(n)$

such that e is relatively prime to $\phi(n)$. Two integers are said to be relatively prime if their greatest common divisor is 1. Since e is relatively prime to $\phi(n)$, $d = e^{-1} \in Z_{\phi(n)}$ exists and can be computed easily using the Euclidean algorithm for finding greatest common divisors. This is because having a multiplicative inverse modulo $\phi(n)$ is equivalent to the existence of an integer d such that $ed \equiv 1 \pmod{\phi(n)}$.

Now the public key is (n, e) and the private key is (n, d) . Actually, there is a symmetric relationship here, so the roles of e and d could be reversed. Thus it makes no difference which key is public and which is private as long as Alice sticks to the choice she makes. The plaintext is represented by an integer between 0 and $n - 1$. How this correspondence between data and integers is made is unimportant so long as there is some unique mapping. Typically, the binary value of the data is the corresponding integer, and the data is broken into blocks of size less than $\lceil \lg n \rceil$. If Alice wants to encrypt P , she computes $E(P) = P^e \pmod{n}$. If she wants to decrypt ciphertext C , she computes $D(C) = C^d \pmod{n}$. It becomes apparent that these are inverse operations by using an extension of Fermat's Little Theorem. The theorem states that if $y \equiv 1 \pmod{\phi(n)}$, then $x^y \equiv x \pmod{n}$. So $(P^e)^d \equiv P^{ed} \equiv P \pmod{n}$.

Notice that successfully implementing RSA means finding random primes within a range, a topic in and of itself, which makes use of primality testing. The problem of primality testing is much easier than factoring, but it often involves the use of probabilistic algorithms. This can lead to a failure of RSA if the primes chosen actually turn out not to be true primes.

2.2.2 ElGamal Cryptosystem

RSA is one of two early types of public key cryptosystems that were successful. RSA is the popular factorization-based algorithm. ElGamal is a popular and easily understood discrete log based cryptosystem. RSA is based on the assumption that factoring is difficult. The discrete log problem refers to finding logarithms in finite groups. This, too, is believed to be a difficult problem. All the information discussed here can be found in [12] with some

examples. (Note: Some basic understanding of finite fields is required for the rest of this subsection).

In the ElGamal system, a finite Field F_q is fixed and a base $g \in F_q^*$ is also fixed, where F_q^* is the cyclic multiplicative subgroup of F_q . It is preferable that g be a generator for the group. Now each user chooses a random integer in $(0, q - 1)$. Let Alice be a user. She must choose a random integer $1 < a < q - 1$ to be her private key. Her public key is then g^a .

Suppose Bob wishes to send a message P to Alice. First, he generates a random integer $1 < k < q - 1$. Then he computes and sends (g^k, Pg^{ak}) to Alice. Now without knowing k , Alice can compute g^{ak} by simply raising g^k to the a -th power. Then Alice divides Pg^{ak} by g^{ak} to reveal P . Division is of course simply multiplying by the inverse which is $g^{q-1-ka} = g^{k(q-1)-ka} = (g^k)^{q-1-a}$ in this simple cyclic group. So in actuality, Alice will never compute g^{ak} but instead $(g^k)^{q-1-a}$. First, we notice that the encrypted message now involves a pair of large integers, making messages approximately twice as long as in RSA. Second, ElGamal is not a deterministic encryption algorithm. Every encryption depends upon random data. This means that identical plaintexts will usually be encrypted to different ciphertexts. Thus, encryption is slowed additionally by the time it takes to generate the random data.

Now if the discrete log problem can be solved, an adversary could compute a and hence decrypt Alice's messages. In theory, there could be a way to compute g^{ak} knowing only g^k and g^a . The Diffie-Hellman assumption is precisely that doing the above is equivalent to solving the discrete log problem. That is, solving one leads to a simple solution of the other. We only know for sure that solving the discrete log problem makes solving the second problem simple. If the Diffie-Hellman conjecture is ever proved correct, then ElGamal will be as strong as the problem of finding discrete logs is difficult.

Implementations and exact definitions of ElGamal vary. Most implementations use a restricted type of finite field, namely fields of the form F_p for some prime p . Also, there is a restriction that $p - 1$ must have at least one large prime factor to be resistant to attacks trying to compute discrete logs over the multiplicative group F_p^* . Where many definitions

vary is in whether to actually require a generator of F_p^* or just an element of high order. The most general definition I have seen is in [24]. In their definition, they just require an element of high order, say m . However, doing this really restricts the algorithm to working over a cyclic subgroup of order m . This means that the key-space is smaller. Now a key has only $m - 2$ possible values. By not using a generator of F_p^* , the system is no stronger than if we would have used F_q^* for some prime q of size similar to m . But our computations are modulo p which is slower than working modulo q . At first glance one might wonder why anyone would work over a subgroup like this instead of finding a generator. The trade-off is that creating the group parameters is quicker this way. It can take significant time to find a generator of F_p^* if p is a random prime. Diffie-Hellman in RSAREF is implemented without finding generator. They first compute a prime q and then find an m such that $p = m \cdot q + 1$ is prime.

[25] gives pseudocode algorithms for finding primes and generators of the multiplicative subgroups. Since verifying a generator requires factoring $p - 1$, randomly choosing a prime p is inefficient. [25] suggests generating a random prime q and then selecting relatively small integers R until we find a $p = 2Rq + 1$ such that p is prime. Several different values q may need to be found before we can find an appropriate p . But once such a p is found, factoring $p - 1$ is quick. Hence, verifying a generator is quick. Our problem with this method is that the probability of finding a generator can be small if $p - 1$ has a lot of small prime factors. Also, the coding is significantly more complex.

A prime of the form $p = 2q + 1$ for some prime q is called a *safe prime* in [25] or a *strong prime* in [24]. These primes are great since there is a 50 percent chance that a random integer is a generator of F_p^* . Also, we need only perform two modular exponentiations to check whether a randomly chosen integer is a generator. We take this a step further and consider all primes of the form $p = 2^k q + 1$ for some small positive integer k . This type of prime has both of the above mentioned properties that a strong prime has. The bonus is that by being less restrictive, they are easier to find. We simply choose a random prime

p . Then we factor $p - 1 = 2^k m$. If m is prime and k not too large, we accept p and find a generator. Actually, we stop early in factoring if k is found to be too large. There is no need to test m in that case. The prime number theorem gives us the ability to compute an approximation to the probability of finding a prime of a given size. Let P_n denote the probability of finding an n bit prime. Then we would expect the probability of finding a prime p such that $p = 2^k q + 1$ for some prime q is at least $(P_n)^2$. This is true if the function that subtracts 1 and factors out the powers of 2 from a number, uniformly distributes its range. If this function somehow avoids primes, then we should notice this empirically. If our conjecture is true and this distribution is uniform among primes and composites, our algorithm for finding suitable primes will be at worst quadratically slower in the average case than just finding a large prime with a large prime factor of $q - 1$.

Thus we see that our method will take more time to generate a suitable prime, but once we do, it is simple to find a generator. In fact, it is expected to take less than two tries. Finding a generator yields stronger security for a fixed prime p . It should be noted that setting up the group parameters only needs to be done once, often for several different users. Rekeying simply involves generating a new random integer and raising the generator to that integer. We use RSAREF's functions for working with large integers, but we implement ElGamal ourselves using only RSAREF's function that tests whether a number is a probable prime.

2.2.3 Elliptic Curve Cryptography

In the past few years *Elliptic Curve Cryptography* (ECC) has gained some popularity, although it has been a significant topic in research for more than a decade. Still, it is not more than half as old as RSA. Part of the recent popularity is due to the smaller key sizes required in ECC. In the world of mobile computing utilizing PDAs, cell phones and smart cards, there is significant interest in algorithms that use less memory or storage. This is in part why ECC has gained recent attention. Secondly, research for new public key cryp-

tosystems was motivated because RSA was under patent for 17 years. Also, companies often support research in ECC and other systems because they do not want to rely completely upon RSA in case it is someday compromised.

Now that we have established why there is interest in ECC, we will discuss its basic concepts. The discrete log problem was discussed with the ElGamal system. There, the discrete log was over a simple cyclic group. However, there is no reason one cannot use another finite group, where inverses are easily calculated. ECC uses a group that is the set of solutions to a particular class of two variable polynomials, called *elliptic curves*. These solutions lie in some field. Before ECC, most work in elliptic curves was about elliptic curves over the complexes, reals or even the rationals. All of these fields are characteristic 0, meaning they are infinite fields. Finite fields have prime characteristics, the characteristic being the size of the prime subfield. So if a field F is of order p^k for some prime p , then we say that F has characteristic p . ECC uses the group of solutions to an elliptic curve over a finite field. Much of the research is focused on using characteristic 2 fields, since they work so nicely with binary and are easier to implement in hardware.

From here on, instead of writing characteristic p , we will use the standard notation of *char* p . We give a formal definition of an elliptic curve below.

Defn: Let K be a field of *char* $p \neq 2, 3$, and let $x^3 + ax + b$ (where $a, b \in K$) be a cubic polynomial with no multiple roots. An *elliptic curve over* K is the set of all points $(x, y) \in K$ such that

$$y^2 = x^3 + ax + b,$$

together with a single element denoted O , called the “point at infinity”.

If K is a *char* 2 field, then an *elliptic curve over* K is the set of points satisfying either

$$y^2 + cy = x^3 + ax + b$$

or

$$y^2 + xy = x^3 + ax^2 + b$$

for some $a, b \in K$ (here we do not care whether the cubic on the right has multiple roots) together with the “point at infinity” O .

If K is a *char* 3 field, then an *elliptic curve over* K is the set of points satisfying the equation

$$y^2 = x^3 + ax^2 + bx + c$$

for some $a, b \in K$ (where the cubic on the right has no multiple roots) together with the “point at infinity” O .

The “point at infinity” mentioned in the above definition turns out to be the identity element of the group. In this group the arithmetic operation is additive rather than multiplicative. So where we wrote g^n in ElGamal, we would write $n \cdot g$ in ECC discussions. In both cases n is an integer and g a group element.

It turns out that many of the signature schemes, cryptosystems and key exchanges based on the discrete log problem in the multiplicative subgroup of finite fields, can be transformed into systems based on elliptic curves. The main difference is that while there is a simple deterministic algorithm to associate plain-text with elements of the multiplicative group of a finite field, there is not any deterministic polynomial time algorithm to do this with elliptic curves. So in practice, very good probabilistic algorithms are used. But using a probabilistic algorithm is nothing new as they are used to generate RSA keys and perform many other number theoretic computations [12].

2.2.4 NTRU

NTRU is a new public-key cryptosystem that has shown up in the past 5 years and has gained some industry support. Having had the opportunity to eat lunch with Joe Silverman, one of NTRU’s designers, I learned what NTRU “unofficial” means: Number Theorist’s R’ Us. We

will not go into great detail about the mathematics behind NTRU here, since it requires a bit of familiarity with ring theory. The algorithm operates over the ring $R = Z[X]/(X^N - 1)$. $Z[x]$ is the ring of all polynomials with integer coefficients. So R is the ring of polynomials with integer coefficients under an equivalence relation that says for any $p(x), q(x) \in Z[x]$, $p(x) \equiv q(x)$ if and only if $p(x) \equiv q(x)$ modulo $(x^N - 1)$. This means that every equivalence class has a representative polynomial of degree less than N . NTRU is also a probabilistic cryptosystem [14]. This means that encryption includes a random element; so each message has many possible encryptions. This is nothing new; we saw that ElGamal works this way as well.

NTRU is not a lattice based cryptosystem, such as the one developed Goldreich, Goldwasser, and Halevi in [13]. However, its success is dependent upon resistance to lattice based attacks. Specifically, security depends on the (experimentally observed) fact that finding extremely short vectors in a lattice is a very difficult problem.

Defn: A *lattice* is a partially ordered set $\langle S, \leq \rangle$ such that for all $x, y \in S$ there exist both a *supremum* (denoted by $x \vee y$) and an *infimum* (denoted by $x \wedge y$).

Let X be a set. An example of a lattice would be the powerset 2^X with the set inclusion relation. The infimum of two subsets would be their intersection, while the supremum of two sets would be their union.

In [14], NTRU is discussed in detail and is compared with RSA in both a theoretical and experimental way. Encryption and decryption speeds take $O(n \log(n))$ operations. This compares with RSA's $O(n^3)$ operations. As a side note, RSA takes $O(n^2)$ operations when small encryption exponents are used. In their experiments NTRU performs 5.9 times faster at encryption, and 14.4 times faster at decryption than RSA. These tests were done on early Pentium 1 CPUs though. Results may differ significantly on modern platforms and hence one should be careful not to read too much into the experimental results. It will be interesting to see how NTRU stands the tests of time, being such a young algorithm. Cryptographers are very skeptical of new algorithms and slow to accept them. This is as it should be, since

the security of all current public key cryptosystems is based on having passed the test of time against thousands of mathematicians examining and trying to break the systems.

2.3 Specialized Video Encryption Algorithms

While we are restricting our API to general block ciphers, it is important to note that there are several specialized video encryption algorithms. These algorithms typically intermix the video encoding algorithms with the encryption. So they are tightly coupled with the particular video format being used.

2.3.1 VEA

Video Encryption Algorithm (VEA) is a specialized video encryption algorithm for MPEG video that was developed within our group and described in detail in [15]. VEA is specialized for video, but it does not intermix the encoding and encryption stages. Encryption occurs on a completely encoded video stream. It makes use of statistical properties of MPEG such as the fact that when viewed as a byte stream instead of a bit stream, the byte values seen are uniformly distributed. This algorithm also makes use of generic block ciphers. Using the statistical properties they discuss, they divide the data in such a way that only half of it need be encrypted. A generic block cipher is used for that. VEA with DES demonstrated a 47% improvement in encryption speed over just using DES. The nice thing about this algorithm is that any generic block cipher will work, and one could expect similar improvements with different block ciphers. Thus our test results (described later) are applicable to those who implement this algorithm as well.

2.3.2 Permutation Algorithm

The permutation algorithm was developed by the same group as VEA and is described in detail in [16]. We developed a known-plaintext attack against it in [17]. In [17] we further

discuss some possible fixes and scenarios in which this algorithm is still quite appropriate.

While the permutation algorithm was developed for video, it can in fact be used on any type of data. One would most likely want to break the data up into chunks roughly the size of a video frame and encrypt each chunk separately. In fact, this algorithm is implemented on a frame by frame basis. The key is a large decimal number perhaps on $O(10)$ digits. This key is used to partition an array the size of the frame in to subblocks. Then the frame is read byte by byte putting data into these different sized subblocks in a round robin manner until all the subblocks are full.

2.3.3 I Frame Encryption

One approach is to encrypt only the I frames of an MPEG video. This clearly entangles the encoding and encryption stages using selective encryption. The encryption is done before the Huffman compression, the final stage of encoding. This actually affects the compression ratio of the video in a negative way. Since the I frames will not have the nice compressible structure of the unencrypted frames, the compression of the I frames will be poor.

This algorithm is discussed in detail in ([18], [19]). In [20] Agi and Gong have shown this method of encryption is insufficient if we desire the video to be completely unusable. Their attack exploits the inter-frame correlation and makes use of the unencrypted I-blocks found in P and B frames.

2.3.4 SECmpeg

SECmpeg, developed by Meyer and Gadegast in [21], is a complete rework of the MPEG format with security in mind. It uses selective encryption and has additional headers, thus making it incompatible with standard MPEG encoders and decoders. SECmpeg can use both DES and RSA. There are four basic security levels set in SECmpeg. Level 1 encrypts only headers. Level 2 encrypts headers and the lower DC and AC coefficients of the I-blocks.

Level 3 encrypts I frames and all I-blocks in P and B frames. Level 4 simply compresses all data, and hence it differs little from using a generic block cipher.

2.3.5 Zig-Zag Permutation Algorithm

The Zig-Zag permutation algorithm in [22] is another selective encryption algorithm. The idea here is to use a permutation of the standard zig-zag ordering of DC and AC coefficients in the DCT transformation of 8x8 blocks in the MPEG or JPEG format. However, Qiao shows in [23] that the Zig-Zag permutation algorithm is vulnerable to a known plaintext attack. This attack exploits the fact that the larger coefficients gather toward the upper-left corner of the macro blocks when in the zig-zag order.

Chapter 3

Related Work

3.1 Cryptographic APIs

We are certainly not the first to develop a cryptographic API. Being the undertaking it is, there must be a good reason to create a new one rather than using an existing API. The original project we were looking at used the Microsoft Crypto API. There are several reasons why simply using this API is insufficient. The obvious reason is *portability*. Even though the project we were working with was developed for Windows, it is mostly portable C++ code. The main hindrance to portability is the use of Microsoft's Crypto API. This was not the biggest issue, though. This API has only a few algorithm choices. This would not be a problem if the API was *expandable*. While the MSDN library provides a very basic outline for creating more Microsoft Cryptographic Service Providers (CSPs), this is not an easy task. It is certainly not a task for a single person, and the only non Microsoft CSP we ever found was developed by Netscape.

While the Crypto++ Library 5.0 [1] overcomes some of the MS Crypto API problems, it is still inappropriate for the task. It has all the algorithms one could ever desire and more. It seems to be frequently updated as well. The problem is that it has an extremely complex interface. It was designed to do so much that the learning curve to use it is overwhelming for someone just needing basic features. After analysis for over two weeks, we were unable to understand it or even extract specific algorithms from the API. So it did not meet our

requirement for simplicity or expandability. However, expandability is less important, since the authors are currently keeping it very up to date.

OpenSSL [2] is an open source distribution of the Secure Socket Layer protocol. The problem is that this is not a general purpose API. It is made specifically for abstracting cryptography away from network communications using a socket like network interface. Its interface is very incompatible with the interface of Distributed Secure Framework for Multicast Transmission (DSFMT) which works by specifically encrypting/decrypting memory buffers. DSFMT is a project developed by our research group for which we would like to remove the dependency on the Microsoft Cryptographic API. The framework's interface would need to be completely changed, and it would become less compatible with different network protocols. DSFMT tries very hard to separate the network and cryptographic functions. Extracting the algorithms from this API to use in another API also proved to be less than worthwhile. It is simpler and more efficient to use other implementations of the desired cryptographic algorithms.

While we want to add cryptographic functions to C++ applications, we note that there is significant cryptography support for Java developers. Java Cryptography Architecture (JCA) [26] is a part of the JDK Security API. The JCA provides functionality for digital signatures and message digests. Because of U.S. export restrictions, the Java Cryptography Extension (JCE) [27] was released separately from the Java 2 platform. The JCE includes support for asymmetric systems, block ciphers, stream ciphers and more. The most important feature of all is that the JCE is a pluggable framework that supports qualified cryptographic providers to be plugged in. This is important because the Sun JCE release is really just a reference implementation to show how the API works. It contains only basic block ciphers such as DES, Triple-DES and Blowfish. [27] contains a link to a list of qualified cryptographic providers that plug into the JCE or completely reimplement it. The Institute for Applied Processing and Communications at Graz University of Technology in Austria has created one of the most comprehensive JCE compliant providers. In fact, they

offer a complete reimplementaion with many modern ciphers. The JCE API is designed to be easily expanded and the sheer volume of qualified commercial cryptographic providers already created for the JCE is evidence of this fact. This is a stark contrast to the Microsoft Cryptographic API which is difficult to expand.

3.2 Cryptographic Algorithm Implementations

The Rijndael algorithm was only recently chosen to be the new Advanced Encryption Standard (AES). This algorithm should be included in any cryptographic API. We surveyed many implementations. We eventually settled upon Brian Gladman's [3] implementation. This implementation was referenced in so many places and was ranked one of the fastest in [4]. Its best competitors were not open source. Gladman's code is portable and available in ANSI C, C++ or x86 assembly. He has good example code, and there is the possibility to use assembly for even higher performance. However, this enhancement would inhibit portability. Speed and clarity ended up being the primary influences on the decision to use his code. The structure of his example code actually had an influence on a great deal of our API code for other algorithms, particularly how he implements CBC mode encryption/decryption.

While AES code is abundant, RC6 code is not. We could find no comparisons between different implementations. Thus, we decided to use Gladman's RC6 code as well since we have experience with his code and reason to trust that it is efficient from prior experience with his Rijndael code. Also, Gladman's design interface is more compatible with our code than the few other alternatives we found.

DES code is abundant and much of it is from the 1980's. However, this should not be surprising since it is old and has been well understood for decades. Unfortunately, many of the implementations were optimized for older platforms. Of the top four coming up in reference, the implementation by Stuart Levy [5] is the most appealing. It is both portable and fast. His code is incorporated into both the DES and Triple-DES classes of our API.

RSA implementations are surprisingly scarce. The Crypto++ Library 5.0 RSA code is difficult to extract. RSAEURO was at one time only allowed outside the US. Though we believe this has changed due to a loosening of cryptographic export laws and the recent expiration of the RSA patent, we could not confirm this. We found an interesting implementation of RSA that is a clever use of C++ classes creating a new class for long integers. It overloads operators to do mathematical operations on these numbers, such as multiplication. The problem is that this code appears very slow, has no key generation code, and it is without author or license information. So while it is very clear and it reveals the mathematical aspect of RSA in a nice way, it is not a good candidate for our API. Instead, the RSAREF 2.0 library is used in our API. It has the same interface as RSAEURO, which is well documented. Also, there is no question of whether it is legal to use. The RSAREF code is not only used for our RSA class, but it is used for creating random bytes in key generation of all the block ciphers. Thus, it has become an integral part of our API. However, anyone wishing to expand the API is in no way bound to RSAREF's use.

ElGamal implementations were also difficult to find. OpenSSL and Crypto++ 5.0 both implement it, but the same problems of extracting other algorithms from those sources apply here. RSAREF implements Diffie-Hellman key-exchange, but not a discrete log based cryptosystem. However, the mathematical functions for dealing with large integers and computations over groups of the form F_p^* , for some large prime p , are in the library. This is because Diffie-Hellman makes use of the same mathematical constructs. Thus by modifying how primes are generated and creating some new structures for ElGamal parameters, we are able to create ElGamal key generation, encryption and decryption functions. The previous chapter on cryptography basics describes the difference in how we generate primes and their predicted implications on performance. Chapter 5 gives real results on performance of our implementation.

Chapter 4

Design and Implementation of Cryptographic Interface

4.1 Overview

4.1.1 Design Goals

Design decisions for our cryptographic API have been influenced by the following goals. The API must be simple for an application programmer to use. It must be simple to extend. Since the set of popular algorithms changes over time (mostly due to advances in cryptanalysis) this API must be easily extensible to include new algorithms. In particular we want the API to be black box upgradeable, meaning that only knowledge of the base class interfaces is needed to extend functionality. A programmer wishing to expand it should not have to understand how other derived classes are implemented to implement their own. However, we provide templates that make adding new classes a simpler task if they choose to use them. The API should be general, only specifying the basic building blocks. This allows its use for many purposes beyond its original design. Lastly, we want an API that can replace use of the Microsoft Cryptographic API with as little recoding as possible.

4.1.2 API Structure

The structure of this API is a hierarchy of key classes. The root of this hierarchy is an abstract key class called the *AKey* class. The purpose of having this abstract class is not for code reuse, but to allow functions to take pointers to this class. This way a function does not need to know what kind of algorithm is associated with the key or even a list of all possible algorithms. It simply uses the encryption and decryption methods declared in this class. So where the Microsoft API will pass handles to key blobs which contain information about the algorithm and the key, users of our API pass pointers to the *AKey* class.

From the *AKey* class we derive two more abstract classes: the *SKey* class for symmetric keys and the *PKey* class for public key cryptosystems. The reason we have these two separate abstract classes is that there are significant differences in symmetric and public key semantics. For example, we just need one export function for symmetric keys, but with public key cryptosystems we may wish to export just a public key, just a private key or both keys. It is a waste of code to recreate an export function for every algorithm. So would like to put it above the actual instances of key classes in the hierarchy. But because a single export function does not work for public key systems, it must be lower than the base abstract class. Creating a symmetric key class provides us a place to put an export function. The export function is just one example of why we need separate subtrees for symmetric and public key cryptosystems.

From the *SKey* class we derive non-abstract classes for each algorithm. These are *AESKey*, *RC6Key*, *DESKey* and *TDESKey* for AES, RC6, DES and Triple-DES, respectively. We also derive non-abstract classes for each of the public key cryptosystems from the *PKey* class. These are *RSAPKey* and *EGKey* for RSA and ElGamal, respectively. When a user creates a new key it will be one of these key classes, and she will most likely allocate it to a pointer for an abstract key class. In the example at the end of this chapter we see this being done in practice.

4.2 Class Details

4.2.1 Base Class: AKey

The base class of the hierarchy is the abstract *AKey* class. This class is a unifying point for the collection of all key classes in the API. Rather than design a function for each type of algorithm or pass the algorithm type to each function, we use pointers to this abstract key class. It declares all of the basic functions of a cryptosystem, including key generation, encryption and decryption. Unless a function needs more specific functions, special to say public key systems, the function can just use a pointer to this abstract key class. If a function is written specifically for public key cryptography, it can require a pointer to a public key class instead.

```
class AKey {
public:
    int GetKeyLen();
    AlgType GetAlgType();
    virtual bool Generate(unsigned long key_len) = 0;
    virtual unsigned long Encrypt(char *buff, unsigned long data_len,
        unsigned long buff_len) = 0;
    virtual unsigned long Decrypt(char *buff, unsigned long data_len,
        unsigned long buff_len) = 0;
    virtual int Import(char *buff, int buff_len) = 0;
protected:
    unsigned long KeyLen;
    AlgType Algorithm;
};
```

The *GetKeyLen* method simply returns the key length. Key length has different meaning for different types of keys. In block ciphers, the key is a simple fixed number of random bytes (Some algorithms must avoid known weak keys such as DES). So for symmetric algorithms this function simply returns the number of bytes in this sequence of random bytes. However, asymmetric keys are much more complex, often including several fields. For example, RSA keys contain a set of large integers. RSAREF includes some extra integers in the key to avoid extraneous recomputation above the basic required integers. Therefore, different implementations may contain different a number of fields. In RSA, the key length is usually known by the size of the modulus in bits where this modulus determines the algebra over which computations are done. So we can see that the definition of key length is specific to the algorithms or algorithm types.

The *Generate* function is used to generate or regenerate keys. In all of the derived classes we have created default key lengths to be passed to this function. It will return *false* only if the key size passed is invalid. Any other type of error is fatal and will cause the program to exit, such as insufficient memory to allocate from the heap.

The *Import* method is used to import key blobs created by the same class. Information about the type of key and algorithm are included at the beginning of the key blob. This would be information such as “this key is a public RSA key pair”. This allows one import function to import private, public keys or even key pairs in the case of asymmetric key classes.

The *Encrypt* and *Decrypt* methods have the same semantics. Both functions return the length of the output ciphertext or plaintext. A zero is returned upon failure. If a null pointer is passed as the input buffer, these functions will return the necessary length needed for the output. *data_len* is the length of input data. *buff_len* is the maximum length of the buffer. Checks are done to make sure that the output will not overflow the buffer. A call with a null pointer can be used first to make sure the buffer is large enough.

4.2.2 Symmetric Key Class: SKey Class

The *SKey* class is an abstract class from which all symmetric key classes are derived. It implements some functions common to all symmetric key algorithms, such as a method to export keys and a method to generate keys based off of a passphrase. It is not usually used by the user directly since its main purpose is for code reuse and structuring.

```
class SKey : public AKey{
public:
    virtual bool PassGen(char *input_buff, unsigned long input_len,
                        long unsigned key_len) = 0;
    int Import (char *buff, int buff_len);
    int Export (char *buff, int buff_len);
    SetMode(Mode mode); //sets the block mode, e.g. ECB, CBC, OFB
protected:
    virtual bool IsValidKeySize(unsigned long size) = 0;
    char *Key;
    Mode CipherMode;
};
```

This class is derived from the *AKey* class. The *Import* function is implemented at this level in the hierarchy. The *Export* function is defined and implemented at this level as well. It has the same syntax as the import function. It also returns a 0 upon failure, but upon success it returns the length of the exported key blob. A NULL pointer argument causes the *Export* method to simply return the required buffer length.

The *PassGen* method is similar to the base class *Generate* method. The main difference is that the *PassGen* method reads data from a buffer to seed key generation. The algorithm to generate a key from a seed is of our own design and makes use of Rijndael encryption.

Evidence that the knowledge of the key does not yield knowledge of the passphrase is in the strength of the Rijndael algorithm. The difficulty of inverting the key generation function is important since a user may use the same passphrase for other systems. This way a key found in plaintext will only compromise systems where that key is used.

The *SetMode* function is specific to block ciphers. It is used to choose between ECB or CBC mode encryption. The information of the mode used is not included in the cipher-text. So the mode must be agreed upon out of band for two parties to communicate. Since the algorithm type, key length and other information is typically agreed upon out of band, it seems only logical to do the same with the mode. Also, it allows for smaller ciphertext. If one does not wish to make out of band communications, the application can always create an application-specific header with all of the algorithm details. For example, in PGP the keys are identified by an e-mail address and a key fingerprint. The only time when there is absolutely no need for communication (out of band or via a header) is the case when the same key and parameters are always used. Here the key can simply be hard-coded into the application. Of course this would have to be a closed source application preferably implemented in tamper-proof hardware.

There is a new protected method added to the *SKey* class. The *IsValidKeySize* takes as input an integer and simply returns a boolean value indicating whether that integer represents a valid key size. For example, this is used by the *Import* method to decide if the key blob is an acceptable size.

4.2.3 Public Key Class: PKey Class

It is from the *PKey* class that all asymmetric or public key cryptosystems are derived. It is the counterpart to the *SKey* class. Some functions unique to asymmetric cryptography are declared here, such as a method that exports entire key pairs. While no code is actually written at this level in the hierarchy, it is a necessary consequence of having a separate symmetric key class. It also unifies the interface of all public key algorithms.

```

class PKey : public AKey{
public:
    virtual int GetMaxInput() = 0;
    virtual int ExportPair (char *buff, int buff_len) = 0;
    virtual int ExportPublic (char *buff, int buff_len) = 0;
    virtual int ExportPrivate (char *buff, int buff_len) = 0;
};

```

The *PKey*, like the *SKey* class, is still abstract. In fact, every method in the *PKey* class is virtual as opposed to the *SKey* class which implements some functions. One function unique to public key algorithms is the *GetMaxInput* method. Unlike block ciphers which all share common modes, such as ECB or CBC, to encrypt messages larger than the block size, public key algorithms have no such standard modes. An application can break up a message into blocks no larger than the maximum input size and mimic the ECB or CBC modes, but this is usually not necessary since public key cryptography is usually performed on small pieces of data such as a symmetric key. Asymmetric algorithms, being so slow, make them impractical for larger encryptions. So the purpose of the *GetMaxInput* method is simply to report this maximum input size and let the application programmer decide how or even if she wants to encrypt larger messages.

There are three types of export functions in this class because often the entire key pair should not be exported. Most frequently only the public key is exported. The *ExportPair* method is not necessary, but in most cases that one wants to export a private key, we find that they want to export both keys. So while not expanding the power of the class, it leads to code simplification for a programmer using the class.

4.2.4 An Example: RC6Key Class

So far, all of the classes we have seen are abstract. Here we look closer at a non-abstract derived class, the *RC6Key* class. The *RC6Key* class implements the RC6 encryption algorithm by Ronald Rivest. It is a symmetric cipher and hence derived from the *SKey* class. Someone creating support for a new symmetric key algorithm would need to create something similar to this class and add it to `key.h`, the main header for the API.

```
class RC6Key : public SKey{
public:
    bool Generate(unsigned long key_len = 32);
    bool PassGen(char *input_buff, unsigned long input_len,
                long unsigned key_len = 32);
    unsigned long Encrypt (char *buff, unsigned long data_len, unsigned
                          long buff_len);
    unsigned long Decrypt (char *buff, unsigned long data_len, unsigned
                          long buff_len);

    RC6Key();
    ~RC6Key();
protected:
    bool IsValidKeySize(unsigned long size);
    bool EncryptBlock(u1byte in_blk[16], u1byte out_blk[16]);
    bool DecryptBlock(u1byte in_blk[16], u1byte out_blk[16]);
    SetKeySchedule();
    u4byte KeySched[44];
};
```

The RC6Key class is a good example of a real derived class of the *SKey* class. One of the first things to notice is that there are some new protected methods and member variables. For instance, *KeySched* is a structure to hold the key schedule. The key schedule is an expansion of the key that describes how to use it in all the different rounds of the algorithm. Key schedules are common to many block ciphers. In many of the algorithm implementations specific structures are used that are unique to the implementation of that algorithm. Since implementations of different algorithms come from different sources, this data could not be declared higher in the *SKey* class. The *SetKeySchedule* method is used to set the schedule for encryption or decryption. In some algorithms the key schedule is different depending upon whether encryption or decryption is being done. The other new protected members of notice are *EncryptBlock* and *DecryptBlock*. Many of the block cipher classes use such methods that work on a single block and are called upon by the public *Encrypt* and *Decrypt* methods.

We also see that there are default arguments to the key generation functions. We have done this in all of the *SKey* derived classes. This is important because the API user may not know what sizes are common or even valid. We have made the default for RC6 256 bits because it suffers no performance loss with larger keys. However, in the AES class we made it 128 bits because larger keys do lead to a performance loss.

4.3 Example API Use

We provide some example code for an application using this code. This is by no means a complete program or even one that compiles. It is simply to demonstrate the syntax. The source code comes with two complete applications, one used for performance testing and one that works much like PGP. Both are heavily documented examples.

First, any application using the code must call the proper headers and declare a key pointer.

```
01: #include ‘‘Key.h’’
```

```

02:
03: void foo(class AKey *Key);
04: void main(int argc, char *argv[]){
05:     class AKey *Key;
06:
07:     Key = new RC6Key();
08:     foo(Key);
09:     delete Key;
10: }

```

Line 1 includes the header that declares all of the classes in the API. Line 3 declares a function which uses a pointer to the base key class. We will describe this function in more detail shortly. We declare a key pointer in line 5. Line 7 is where we actually create a key and assign it to a pointer to the base key class. In particular this is an RC6 key that we create here. Line 9 cleans up the key and erases sensitive information from memory. Line 8 calls a function *foo* which uses a pointer to the abstract base class. Because *RC6Key* is inherited from the *AKey* class, we were able to make the assignment in line 7 and hence pass an RC6 key to *foo* in line 8. Let us look closer at what *foo* actually does.

```

01: void foo(class AKey *Key){
02:     char input[80];
03:     char *output;
04:     int output_len;
05:
06:     Key->Generate(24);
07:
08:     output_len = Key->Encrypt(NULL, 80, 80);
09:     output = new char[output_len];

```

```
10:     memcpy(output, input, 80);
11:     output_len = Key->Encrypt(output, 80, output_len);
12: }
```

Line 6 generates a key. The argument is the key length in bytes. So we have chosen a 196 bit key in this instance. If no argument would have been used, a default length would have been used. Line 8 is very important. Because the encrypted result may be longer than the plaintext, calling the *Encrypt* method with a NULL pointer returns the length needed for a buffer to hold the result. Lines 9 and 10 allocate the buffer and copy the input to it. Line 11 actually performs the encryption since a non-NULL pointer to the output buffer is used. The encryption function double checks that this buffer is long enough. We avoided any problem by making sure to allocate a long enough buffer ahead of time. The *Encrypt* method returns the length of the actual output. This could potentially be smaller than the buffer length. This is because there may be a cryptographic algorithm where the output length is dependent on more than just the input length. For such an algorithm line 8 would return a maximum length of the output. In the case of block ciphers, the output length is completely determined by the input length. So in our example line 11, would not change the value of *output_len*.

Chapter 5

Performance Results

5.1 Overview

5.1.1 Platforms

Performance tests have been made on three different computers. The first platform is a Pentium III 600 MHz with a 133 MHz bus speed. It has 384 MB of RAM and runs Windows 2000 Workstation. The second computer is an Athlon XP 2000+ with 2 GB of RAM. It runs Windows XP Professional edition and is a moderately loaded file server. Tests were run when no one else was accessing the machine, though. The Athlon is actually a dual processor machine, but since the test software is single threaded, it has little effect on performance. The program is scheduled to a single CPU. The only effect the other processor may have is that the load on the current CPU may be lightened. The third machine is a Pentium IV 2.0 GHz which is supposed to be comparable to the Athlon XP 2000+. It has 1 GB of RAM and runs Windows XP professional.

5.1.2 Block Cipher Tests

We examined AES and RC6 with key sizes of 128, 196 and 256 bits each. We also examined DES (56 bit key) and Triple-DES (168 bit key). We learned that key generation runs in almost constant time depending only upon the key length. We saw that encryption

and decryption performed similarly. We also found that some algorithms are optimized for particular platforms. More surprisingly, some algorithms run at the same speed regardless of key length and depend only upon the platform. Lastly, we examined the performance implications of using CBC mode instead of ECB mode.

5.1.3 RSA Tests

We ran tests on key generation, encryption and decryption for RSA. The encryption tests were performed on large key blobs for a symmetric cipher since that is generally what one encrypts with public key cryptography. We saw that the exponential time required for key generation is very costly compared to the cubic time required for encryption and decryption. Also, we found that there is a constant factor difference in performance between encryption and decryption, resulting in a difference of orders of magnitude between the two operations. So where encryption may take tens of milliseconds, decryption can take seconds, and key generation may even take minutes.

5.1.4 ElGamal Tests

We ran the same sort of tests on ElGamal as on RSA. We found the same cubic behavior for encryption and decryption, however the operations are on the same order of magnitude. The key difference we found is that key generation is much quicker for ElGamal, taking even less time than encryption or decryption. The caveat is that group parameter generation, done once per PKI, is extremely expensive.

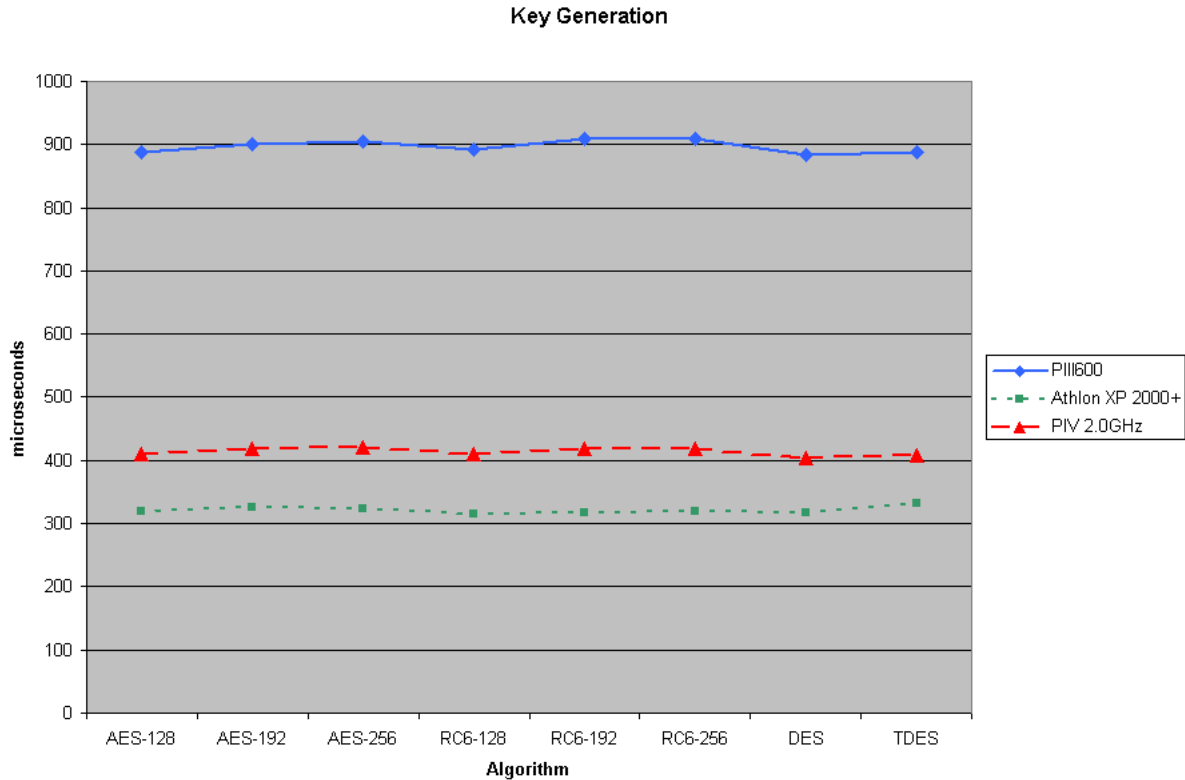


Figure 5.1: Here we plotted the key generation time in microseconds for different algorithms. AES and RC6 have both been tested with 128, 196 and 256 bit keys. DES always uses a 56 bit key, and Triple-DES was implemented with a 168 bit key. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test.

5.2 Block Cipher Comparisons

5.2.1 Key Generation

Figure 1 shows that the Athlon generates keys the fastest, but key generation on any of the platforms is still less than 1 millisecond. We can also see that while key generation speed is linear with respect to key size, the cost of key generation is dominated by a very large overhead. This overhead is from calling a system timer repeatedly to seed the RSAREF random structure. Once this random structure is seeded, the algorithm quickly generates pseudo-random bytes. Since the key generation code for all the algorithms is so similar, the algorithm choice has little impact. DES and TDES are slowed down slightly because checks

for weak keys are done when they are generated.

5.2.2 Encryption Performance

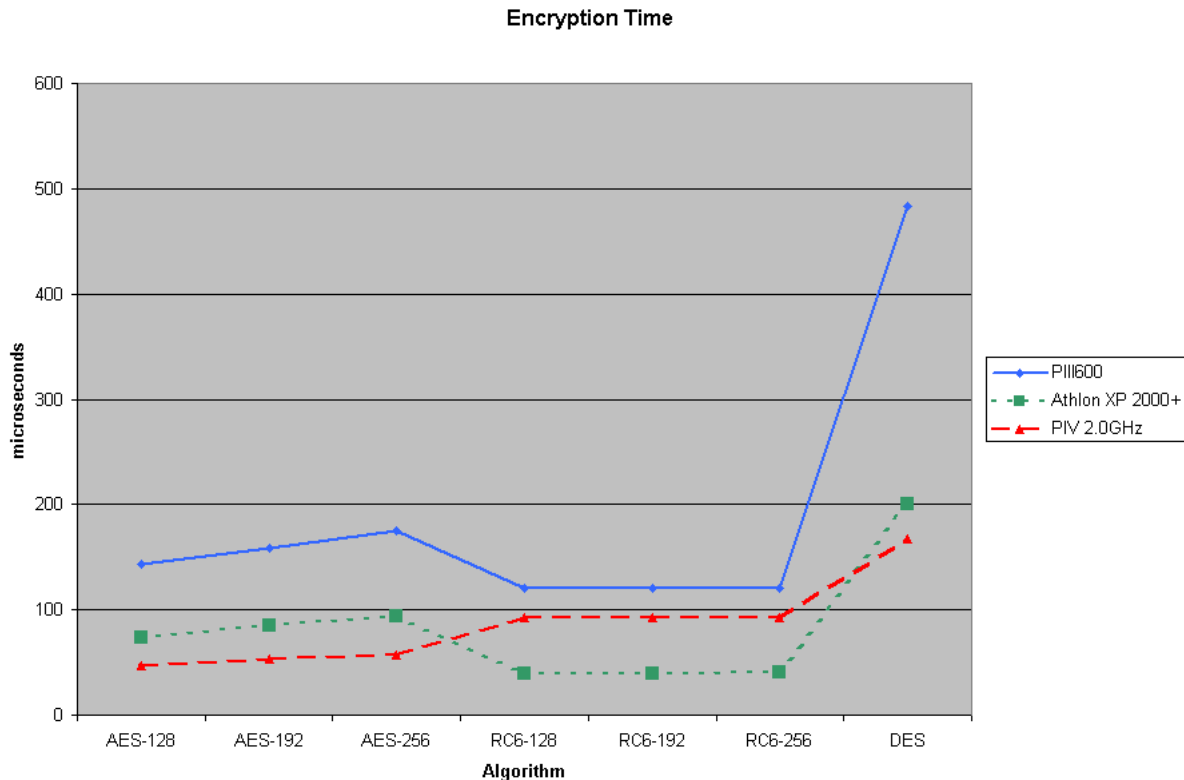


Figure 5.2: Here we plotted the encryption time in microseconds for different algorithms. AES and RC6 have both been tested with 128, 196 and 256 bit keys. DES always uses a 56 bit key, and Triple-DES was implemented with a 168 bit key. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test.

Figure 2 gives us a great deal of information about encryption performance. All of these times come from encryption of a 1500B message, the size of an Ethernet packet. They are averaged over 3 runs of 100 iterations each. Each round of tests gave results with very little deviation from the mean.

The first obvious conclusion is that AES and RC6 are very fast. TDES is not even on the chart since it is so slow. AES and RC6 take less than 200 microseconds, even on the slowest computer with the slowest algorithm. The next important thing we see

is that performance is constant across key sizes for RC6, while it increases linearly for Rijndael. Perhaps surprisingly, we find that the fastest algorithm depends upon platform. It is noted in [7] that RC6 performs much better on the Pentium II and III processors than RISC processors. This is because of the amount of data dependent rotations and 32 bit multiplications that RC6 uses. The RISC architectures are not as fast at these operations. We also see that the Pentium IV is not optimized for these operations. On both the Pentium III and Athlon, RC6 is faster than AES, but this is reversed on the Pentium IV. However the fastest result of all comes from RC6 on the Athlon, reaching an encryption time as low as 39 microseconds. This translates to a throughput of over 307Mbps on a general purpose CPU.

5.2.3 Decryption Performance

Figure 3 shows the results of decryption tests. All the comments on encryption apply here. Also, we see that with RC6 and DES, encryption and decryption speed are the same. However, decryption is slightly slower than encryption with AES.

5.2.4 ECB vs CBC

CBC mode, described in section 2.1.9, costs a fixed amount of extra time per block in extra memory copies and **XOR** operations. These are independent of the choice of encryption algorithm. So one would expect that CBC mode would be a little slower and that this effect would be less noticeable with slower algorithms. A preliminary test with RC6-256 and 1500B packets showed that CBC mode takes 8.3 times as long. This was very surprising. We soon realized the cause was the constant overhead of generating an *Initialization Vector* (IV). We saw key generation is dominated by the cost of initializing the RSAREF random structure in a previous section. This made us realize what was making CBC slow, especially when encrypting a small message; it was the cost of creating an IV. By using Gladman's code for

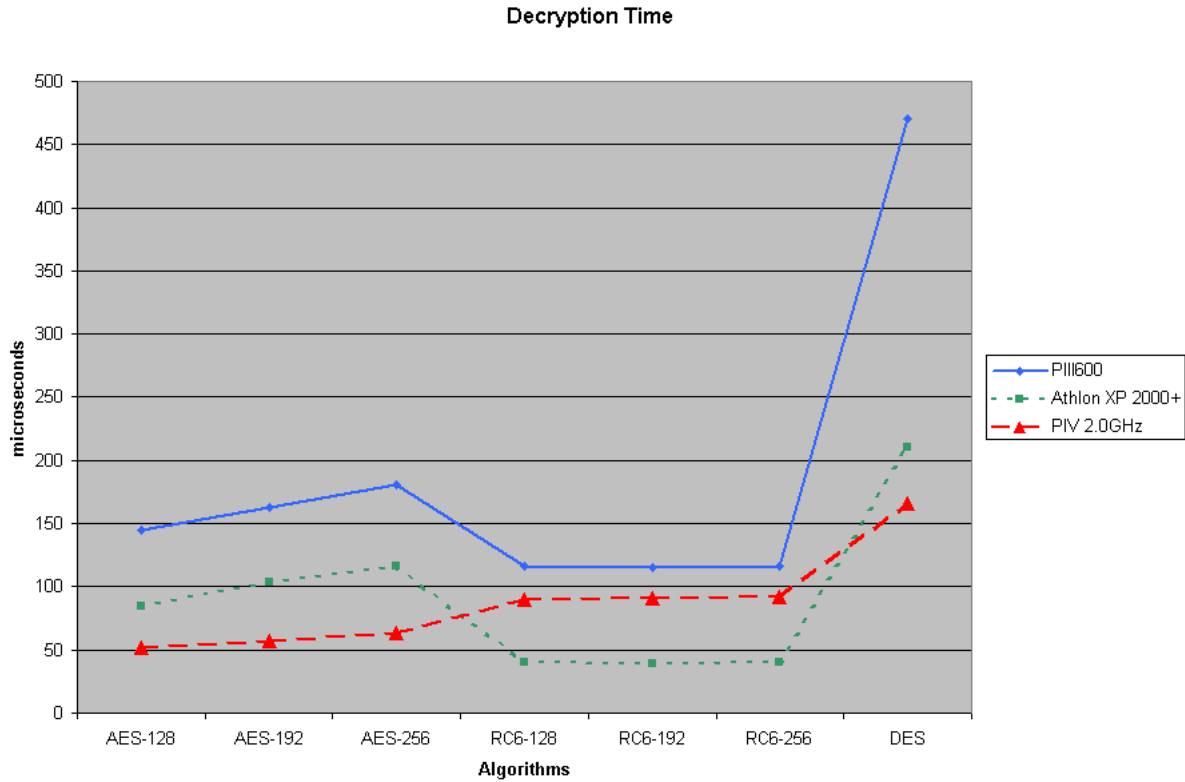


Figure 5.3: Here we plotted the decryption time in microseconds for different algorithms. AES and RC6 have both been tested with 128, 196 and 256 bit keys. DES always uses a 56 bit key, and Triple-DES was implemented with a 168 bit key. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test.

IV generation, which is less random, performance of CBC was only 80% slower. This made a huge difference.

Tables 1, 2, and 3 compare performance of ECB vs CBC on the Athlon XP 2000+ platform. This is done for AES-128, RC6-128 and TDES. This data confirms several predictions. The first obvious fact confirmed is that ECB is faster than CBC. We also see that as the message sizes get larger, the throughput becomes greater. This is because the effect of the

| Data Size | Throughput (Mbps) ECB/CBC | %Slower |
|----------------|---------------------------|---------|
| Packet - 1500B | 164/110 | 67.1 |
| DTV - 5556B | 196/138 | 70.4 |
| HDTV -42KB | 210/148 | 70.5 |

Table 5.2: Encryption - RC6-128, Athlon XP 2000+

| Data Size | Throughput (Mbps) ECB/CBC | %Slower |
|----------------|---------------------------|---------|
| Packet - 1500B | 308/162 | 52.6 |
| DTV - 5556B | 367/202 | 55.0 |
| HDTV -42KB | 390/216 | 55.4 |

Table 5.3: Encryption - Triple-DES, Athlon XP 2000+

| Data Size | Throughput (Mbps) ECB/CBC | %Slower |
|----------------|---------------------------|---------|
| Packet - 1500B | 20.7/19.3 | 93.2 |
| DTV - 5556B | 20.8/19.9 | 95.7 |
| HDTV -42KB | 21.3/20.2 | 94.8 |

one time costs for setting the key schedule, extra cipher-text stealing operations and IV generation are spread out over a larger message. Also, we see that the ratio between ECB and CBC throughput stabilizes as the message size grows. This again is a result of the limited influence of an initial cost, namely IV generation. This ratio's limit behavior allows one to calculate the amount of extra time spent per block in CBC mode. We see that the effect of this ratio is more pronounced with faster algorithms just as we predicted. In particular we can see that it is .554 to 1 for RC6 where it is .948 to 1 for the very slow Triple-DES. This is because the extra cost per block for CBC is independent of the algorithm choice.

One odd thing is that the ratio of ECB to CBC throughput for Triple-DES is not strictly increasing as message size grows. This is most likely because the effect is small enough for such a slow algorithm that normal variances between test runs can cause enough error to disturb the strictly increasing behavior. The fact that all of the ratios are so close together verify that the behavior we expect to observe is very sensitive to error.

5.3 RSA Performance

Figures 4, 5 and 6 show the data gathered from tests of RSA. All encryption and decryption has been performed on a 320 bit key blob, the size of a key blob for a 256 bit key. This is the size of the largest symmetric key blob in our API. Key generation tests were only repeated

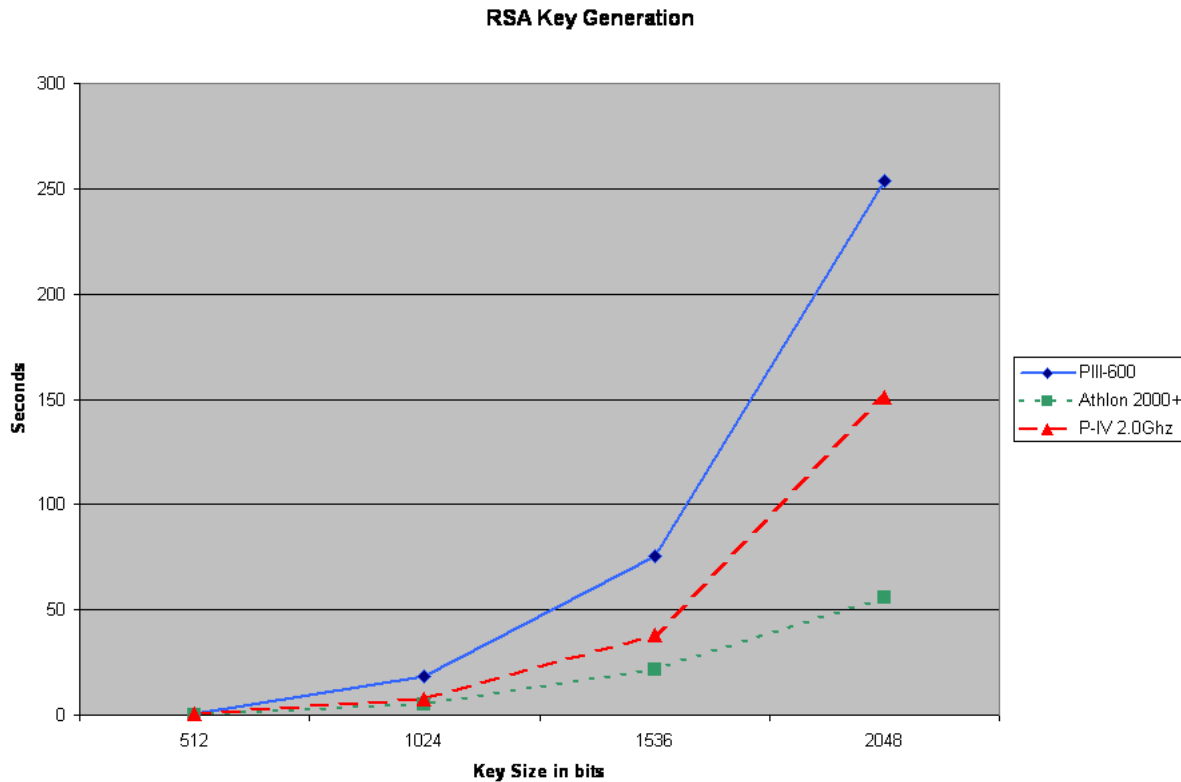


Figure 5.4: Here we plotted the key generation time in seconds for different RSA key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 25 iterations per test.

75 times, whereas encryption and decryption tests were computed 300 in the runs of 100 iterations each. The key generation tests were fewer because of the inordinate amount of time it takes to generate large keys. Tests were done on the same three platforms as the symmetric algorithm tests: a Pentium III 600 MHz, Athlon XP 2000+ and a Pentium IV 2.0GHz.

The first observation is that RSA is very expensive in the cost of time. Encryption is more than three orders of magnitude slower than that of symmetric key cryptography. Key generation is six orders of magnitude slower than that of symmetric key generation. Also, the cost of time for key generation, encryption and decryption all increase super-linearly with respect to key size. More specifically, encryption and decryption speed is cubically related to the key size, and key generation is exponentially related to the key size. This super-



Figure 5.5: Here we plotted the encryption time in milliseconds for different RSA key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test.

linear relationship is especially apparent from the key generation and decryption graphs. We also see orders of magnitude difference in the speeds of encryption versus decryption. This is not all bad, though. In a multicast system the key server has many encryptions per decryption that a client has. In other words, decryption costs are spread across the many client machines.

A surprising result is that there seems to be a larger than expected performance increase from using the Athlon over the Pentium 4. For key generation the Pentium 4 performs about half way between the Pentium III and the Athlon, even though the Pentium IV is supposed to be almost as fast as the Athlon. This indicates that there are certain operations used in RSA that are not as optimized on the Pentium IV architecture, just as in the case of the RC6 algorithm.

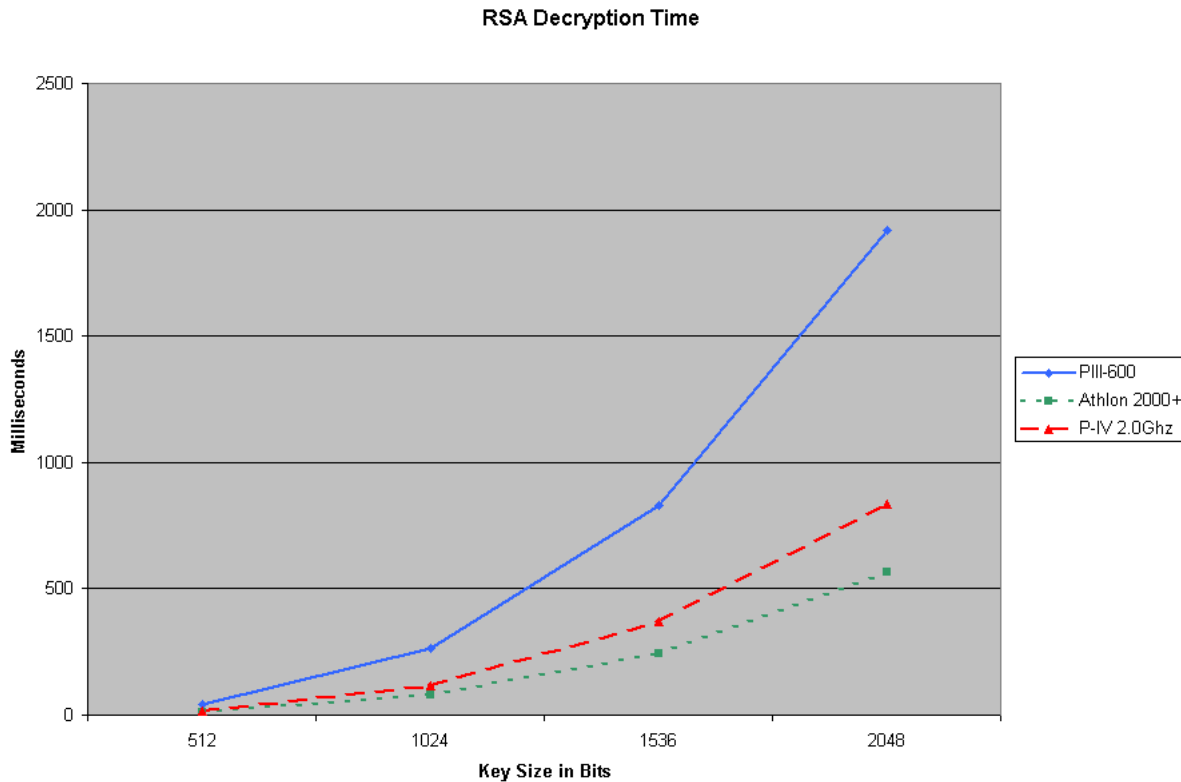


Figure 5.6: Here we plotted the decryption time in milliseconds for different RSA key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test.

The decision of a good key size is especially important when the costs grow cubically with key size such as they do with RSA. A very general principle upon the current security of different RSA key sizes follows. 512 bits is considered weak in modern times and should not be used for anything very permanent. It is sufficient to keep a normal adversary at bay, but it is inappropriate for professional use. 1024 bits is considered OK, and really it is recommended that one does not go below this. This is actually what is used in most current cryptosystems and for most certificates. 1536 bit keys are considered to be good. They come at a significant cost of performance, but they are a good alternative that falls between 1024 bit and 2048 bit keys. 2048 bit keys are considered very strong. I have not seen anything larger used. This is the size I use to sign my e-mails. As long as keys are not being generated frequently and a machine is not acting as a server doing a lot of encryption, the cost is still

not very significant. However, one might not want to use such a large key in a multicast system.

5.4 ElGamal Performance

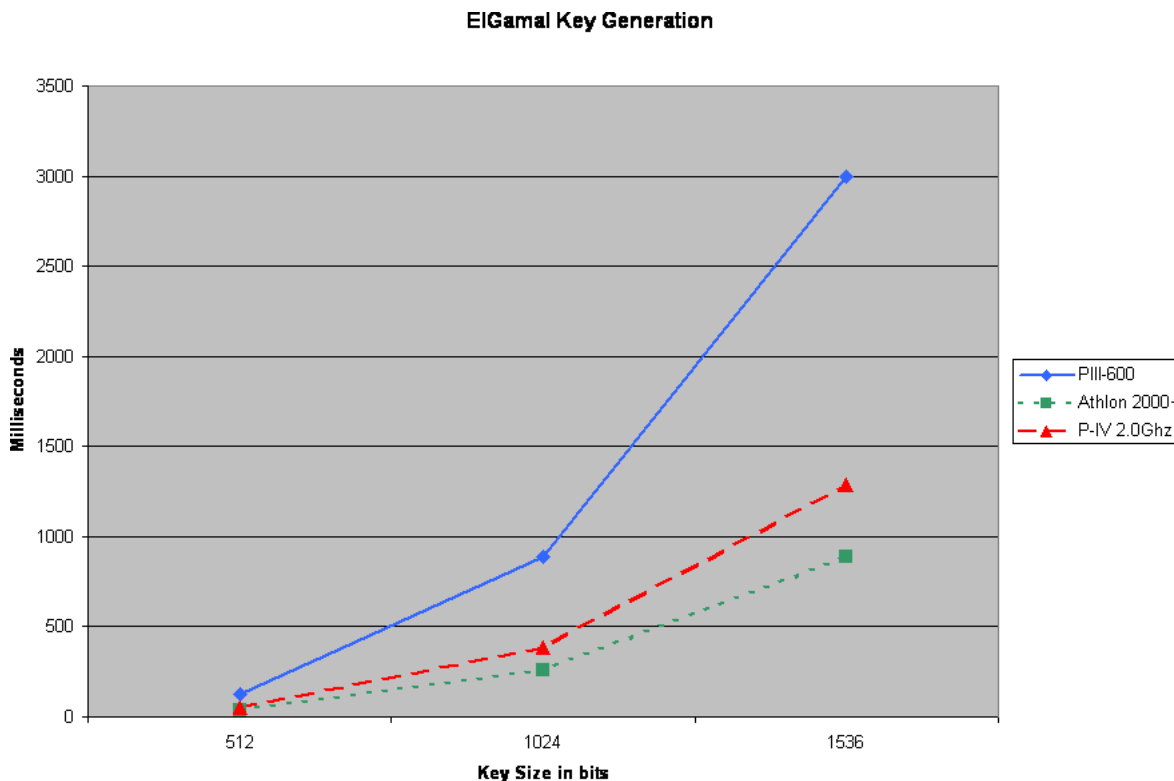


Figure 5.7: Here we plotted the key generation time in milliseconds for different ElGamal key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test.

The ElGamal implementation uses the same package to compute operations on large numbers as our RSA implementation. Also, the mathematical operations are similar in that they both use a lot of modular arithmetic including addition, multiplication and exponentiation modulo a large integer. Because of these similarities, we would expect to see the same cubic behavior in the encryption and decryption tests, and we do. Though this may be trivial to say since we only have three data points which can be made to fit even a quadratic polynomial. But the point is that we see a similar super-linear behavior that works on the

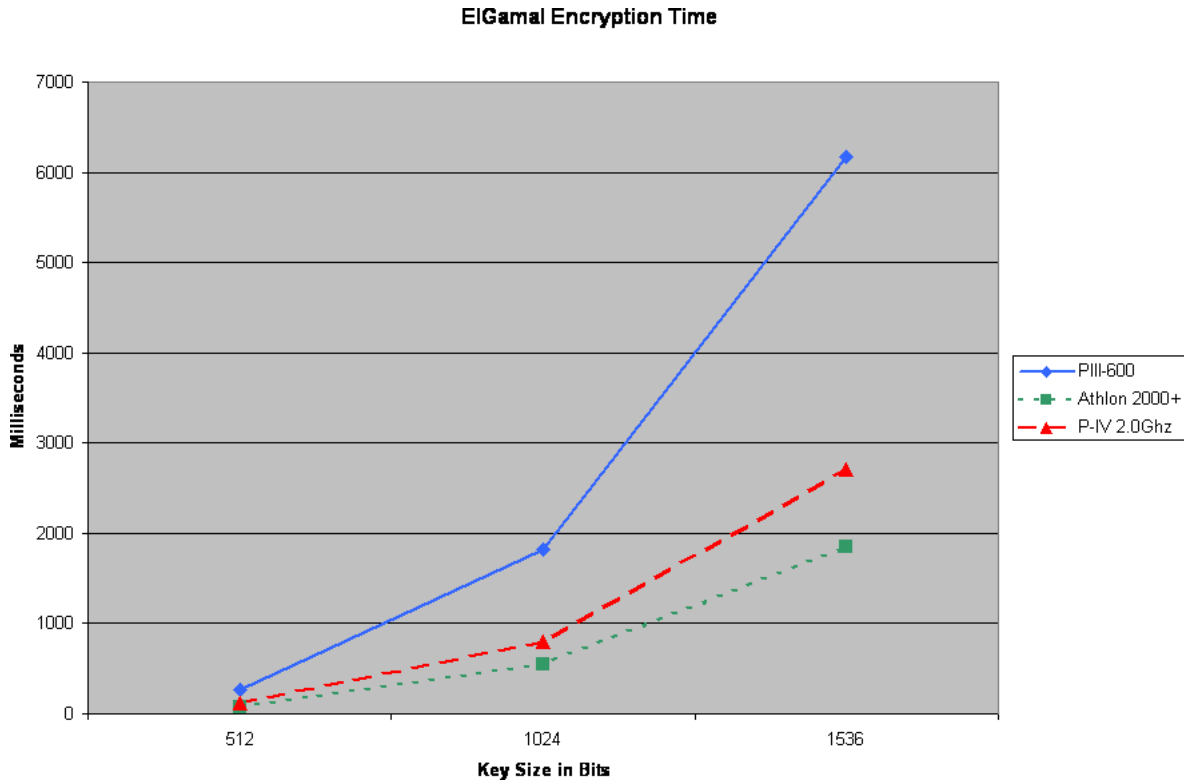


Figure 5.8: Here we plotted the encryption time in milliseconds for different ElGamal key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test.

same order of magnitude as RSA. Interestingly, all three tests (key generation, encryption and decryption) can be made to look identical to the others by multiplying the results by the appropriate constant. The slowest test results, i.e. the cubic with the largest constant, is encryption which has the most operations in the source code. The fastest test results are for key generation which has the fewest operations in the source code. So these constants are strongly related to the number of large integer computations.

Immediately, we see a big difference from RSA in key generation. Key generation takes an exponential amount of time with RSA, but key generation runs faster than encryption and decryption with ElGamal. The reason is important. With RSA, we had to generate two large primes each half the size of the final modulus. It gets exponentially harder to find larger primes and hence the exponential behavior observed with key generation. In ElGamal, all that is done is choosing a random integer within given bounds and raising

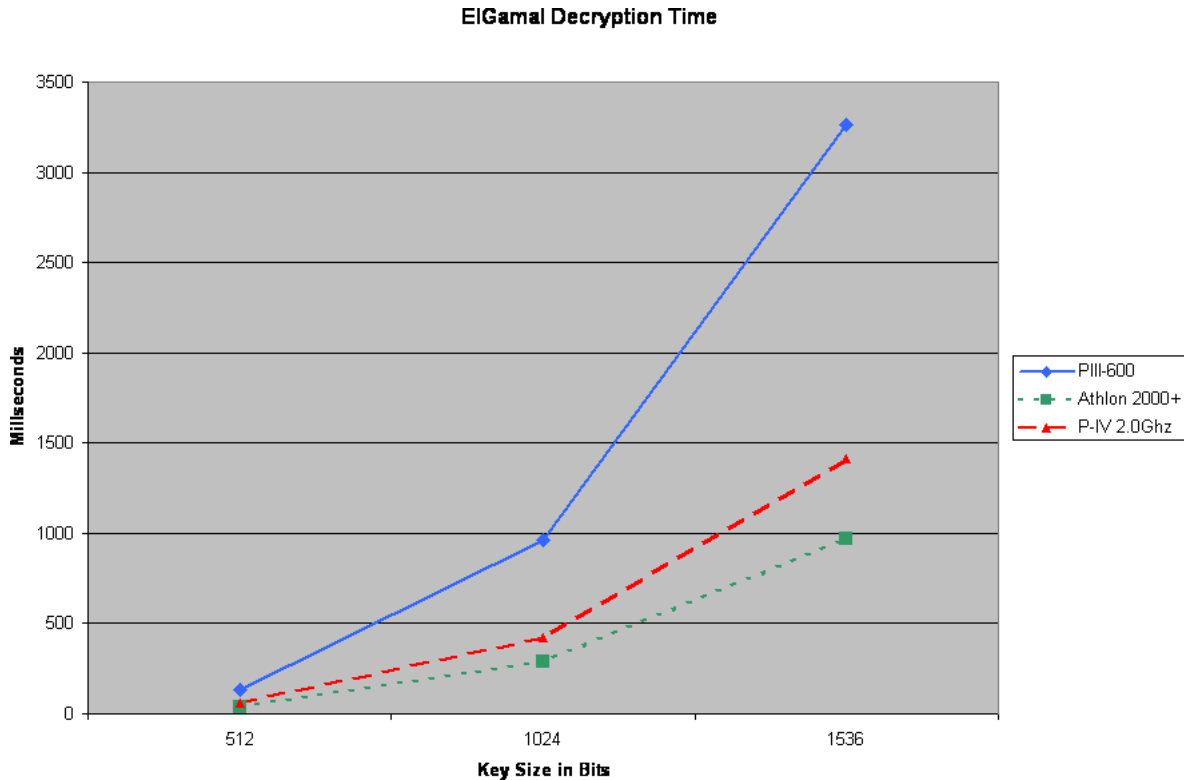


Figure 5.9: Here we plotted the decryption time in milliseconds for different ElGamal key lengths listed in bits. There is a line graph for each of the three test platforms: Pentium III 600MHz, Athlon XP 2000+, and P-IV 2.0GHz. Each test ran three times with 100 iterations per test.

the generator to that power. These are the sort of operations performed while encrypting and decrypting, and that is why key generation happens on the same order of time. Now this does not mean that finding primes is not important in ElGamal, but they only need to be computed once for an entire PKI. The prime chosen is always made public and is not weakened by an entire organization using the same prime. Individuals can use different keys which are essentially random integers greater than 1 and less than $p - 1$, where p is the prime. Individuals could even choose different generators if they like, for extra security in case an attacker tried making a discrete log look-up table for the generator. But this isn't really necessary since the storage for such a table is unattainable for a reasonable key size.

Choosing the prime to use for an organization does take a while, though. We noted earlier that the primes we use for ElGamal are strong primes and are expected to occur with a probability that is the square of the probability for a normal prime of a given length.

Also, the prime in a 1024 bit key, for example, must be 1024 bits. But for RSA, a 1024 bit key only requires two 512 bit primes. Let $r(n)$ be a function representing the time taken to generate an n bit RSA key. Then we expect an ElGamal key to take roughly $\frac{1}{2}(r(2n))^2$ time. Because this is so slow, we only made keys up to 1536 bits and on the fastest platform, the Athlon 2000+. On this platform it took 128.75s to generate a 512 bit key, 5.06 hours for a 1024 bit key and 45.99 hours for a 1536 bit key! We must keep in mind two things, though. First, this is done once for many keys. Second, the discrete log problem for an n bit prime is significantly harder than factorization of a product of two primes of length $\frac{n}{2}$ bits each. So security for a 1024 bit ElGamal system is above normal needs, and once the first prime has been generated, key generation is done faster than encryption and decryption operations. This makes ElGamal a good alternative to RSA.

Since the operations are mathematically similar to those in RSA, we see the same performance comparisons between platforms. The Pentium III is the slowest, and the Athlon is the fastest again. The random data generation needed for encryption and key generation is not significant since it grows linearly, and with a very small slope as we saw when we examined random data generation more closely in the context of block ciphers. If it were very significant, we would expect to see decryption performing better than key generation. It would also mean that encryption and decryption times would not be related as simply as multiplying by a constant.

Chapter 6

Conclusion

This design of our cryptographic API stemmed from a need to expand the algorithm selection of the Microsoft Crypto API. We have shown that the API has a simple interface. This interface provides simple guidelines for an application programmer to extend the API, giving her complete freedom in the details of the implementation of new algorithms. We have provided implementations of AES, DES, Triple-DES, RC6, RSA and ElGamal. There are templates for use if one decides to add classes for more algorithms. A PGP like application to encrypt files is provided to test any new functionality added to the API which can be compiled as a Windows DLL or a Linux .so. The API is powerful because basic primitives are provided. This allows the application programmer to create more complex cryptographic functions, such as digital signatures, by using the basic methods provided. The API hides all details of specific encryption algorithms while presenting a uniform interface. So while the user of the API should know basic cryptographic concepts, she will not need to be a mathematician. Lastly, much of the function syntax follows that of the Microsoft Cryptographic API. This means that this API can replace use of the Microsoft Crypto API with minimal code change.

We have presented performance results of this API on three different x86 architectures using a variety of key sizes and cipher modes. It has been shown that architecture influences performance of specific algorithms greatly. We have used the data to verify different hypotheses concerning what parts of the algorithm take the most time and to understand asymptotic behaviors. For instance, we showed that generation of random data for IVs or keys can easily

dominate the time taken when messages are short or encryption algorithms are fast. Also, it was shown that key generation is extremely expensive in RSA, and encryption and decryption are several orders of magnitude slower than the block ciphers. Additionally, the costs of increasing keys sizes is cubic for RSA, while linear at worst for the block ciphers. ElGamal was shown to perform similarly to RSA except with faster key generation. We have even shown that new algorithms such as AES perform far faster than DES which is tremendously weaker. It was also shown that while Triple-DES is still the most common block cipher used, it performs so poorly that it is not even on the charts with RC6 or AES.

Chapter 7

Future Work

While our crypto API is stable, one may want to implement some alternatives to RSA and ElGamal. Right now these are the only public key algorithms available for use in the API. A newer, but popular, cryptosystem is *Elliptic Curve Cryptography* (ECC). One common ECC algorithm is a variant of ElGamal with the difference being that the underlying algebra is no longer the multiplicative subgroup of a finite field, but it is the set of solutions to an elliptic curve over a finite field. ECC keys are much smaller than RSA keys. Thus ECC is a often a good alternative for mobile devices. *Hyperelliptic Curve Cryptography* (HCC) and NTRU are newer cryptosystems yet, and they may prove worthwhile to add to the API.

Another important addition to this API would be methods for signing and verifying. These methods would be added to the *PKey* class since signing semantics are only clear for public-key cryptosystems. One could create a new class for hashes to do this. Hashes are needed to create digital signatures, and they are used for integrity checking as well. Thus an abstract class for hashes may prove very useful. MD5, SHA-1, SHA-2 and RIPE-160 are all good candidates for inherited hash classes.

While we have implemented block ciphers, one may find a desire to use stream ciphers. They could do this by creating a cipher class inherited from the *AKey* class. The symmetric key class is really made for block ciphers. Though, one can always use a block cipher in place of a stream cipher by using OFB or CFB modes. However, a stream cipher cannot as easily be made to replace a block cipher. So one might consider adding OFB and CFB modes to

the *SKey* class before creating a new stream cipher class.

Right now this API is available as a Windows DLL created in Microsoft Visual Studio and a Linux .so created with gcc. If someone wishes to port the code to another UNIX system, they would likely only need to make some minor makefile changes. The only other change that may need to be done is some adjustments to deal with big-endian architectures in the low level coding of the algorithms. Anyone wishing to port to a big-endian architecture such as a Sun Solaris, should definitely investigate the need for such changes.

References

- [1] Wei Dai. *Crypto++ Library 5.0*. At <http://www.eskimo.com/~weidai/cryptlib.html>
- [2] *OpenSSL*. At <http://www.openssl.org>
- [3] Brian Gladman. *AES Source Code Implementation*. At http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm
- [4] Helger Lipmaa. *A Survey of Rijndael Implementations*. At <http://www.tcs.hut.fi/~helger/aes/rijndael.html>
- [5] Stuart Levy, Minnesota Supercomputer Center. *FastDes Source Code*. April 1988
- [6] Bruce Schneier. *Applied Cryptography - Second Edition*. 1996
- [7] M.J.B Robshaw. *RC6 and the AES*. January 2001 Available at <http://www.rsasecurity.com/rsalabs/rc6/>
- [8] Susan Landau. *Designing Cryptography for a New Century*, Communications of the Association for Computing Machinery, Vol. 43, No. 5, May 2000, pp. 115–120.
- [9] Bruce Schneier and Doug Whiting. *Performance Comparison of the Five AES Finalists*. AES Candidate Conference 2000: 123-135
- [10] Abdul Hamid M. Ragib, Nabil A. Ismaili, and Osama S. Farag Allah. *Enhancements and Implementation of RC6 Block Cipher for Data Security*. IEEE Catalog Number: 01CH37239, Published 2001.
- [11] C Adams, H.M. Heys, S.E. Tavares, and M. Wiener. *An Analysis of the CAST-256 Cipher*. Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering, 1999.
- [12] Neal Koblitz. *A Course in Number Theory and Cryptography - Second Edition*. Graduate Texts in Mathematics 114, Springer-Verlag, 1994
- [13] O. Goldreich, S. Goldwasser, S. Halvei. *Public-key Cryptosystem from the Lattice Reduction Problems*. MIT - Laboratory for Computer Science, preprint November 1996
- [14] Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman. *NTRU: A New High Speed (Ring-Based) Public Key Cryptosystem*. Algorithmic Number Theory - ANTS III proceeding , pp. 267- 288, Springer-Verlag, 1998

- [15] Lintian Qiao and Klara Nahrstedt. *A New Algorithm for MPEG Video Encryption*. In Proceedings of The First International Conference on Imaging Science, Systems, and Technology (CISST'97), pages 21-29, Las Vegas, Nevada, July 1997. Available via www at <http://cairo.cs.uiuc.edu/publications/paper-files/mpg-security.ps>
- [16] Hao-hua Chu, Lintian Qiao, and Klara Nahrstedt. *A Secure Multicast Protocol with Copyright Protection* in Proceedings of IS&T/SPIE's Symposium on Electronic Imaging: Science and Technology, San Jose, CA, January 1999.
- [17] Adam Slagell. *Known Plaintext Attack Against a Permutation Based Video Encryption Algorithm*, 2002 Available at <http://www.slagellware.com>
- [18] Y. Li, Z. Chen, S. Tan, and R. Campbell. *Security enhanced MPEG player*. In proceedings of IEEE First International Workshop on Multimedia Software Development (MMSD'96), Berlin, Germany. March 1996
- [19] T.B. Maples and G.A. Spanos. *Performance Study of Selective Encryption Scheme for the Security of Networked, Real-time Video*. In Proceedings of 4th International Conference on Computer Communications and Networks, Las Vegas, Nevada. September 1995
- [20] I. Agi and L. Gong. *An Empirical Study of MPEG Video Transmissions*. In Proceedings of the Internet Society Symposium on Network and Distributed System Security, pages 137-144, San Diego, CA. February 1996
- [21] J. Meyer and F. Gadegast. *Security mechanisms for multimedia data with the example MPEG-1 video*, 1995 Available on www at <http://www.powerweb.de/phade/phade.html>
- [22] L. Tang. *Methods for Encrypting and Decrypting MPEG Video Data Efficiently*. In Proceedings of the Fourth ACM International Multimedia Conference (ACM Multimedia '96), pages 219-230, Boston, MA. November 1996
- [23] Lintian Qiao, Klara Nahrstedt and Ivan Tam. *Is MPEG Encryption by Using Random List Instead of ZigZag Order Secure?*, 1997 Available via www <http://cairo.cs.uiuc.edu/publications/paper-files/isce97.ps>
- [24] Yiannis Tsiounis and Moti Yung. *On the Security of ElGamal-Based Encryption*. Lecture Notes in Computer Science, (vol. 1431, pg 117), 1998. Available at <http://citeseer.nj.nec.com/tsiounis98security.html>
- [25] Alfred J. Menezes, Paul C. Van Oorschot and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996
- [26] *Java Cryptographic Architecture - API Specification and Reference*. Available at <http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html>

- [27] *Java Cryptographic Extension*. Available at <http://java.sun.com/products/jce/index-122.html>
- [28] *IACK JCE Implementation*. Institute for Applied Information Processing and Communications at Graz University of Technology. Available at <http://jce.iaik.tugraz.at>