2014-06-01

A New Public-Key Cryptosystem

Christopher James Hettinger
Brigham Young University - Provo

A New Public-Key Cryptosystem

Chris Hettinger

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Paul Jenkins, Chair
Darrin Doud
Pace Nielsen

Department of Mathematics

Brigham Young University

June 2014

ABSTRACT

A New Public-Key Cryptosystem

Chris Hettinger
Department of Mathematics, BYU
Master of Science

Public key cryptosystems offer important advantages over symmetric methods, but the most important such systems rely on the difficulty of integer factorization (or the related discrete logarithm problem). Advances in quantum computing threaten to render such systems useless. In addition, public-key systems tend to be slower than symmetric systems because of their use of number-theoretic algorithms. I propose a new public key system which may be secure against both classical and quantum attacks, while remaining simple and very fast. The system's action is best described in terms of linear algebra, while its security is more naturally explained in the context of graph theory.

Keywords: Cryptography, Public Key, Post-Quantum

Acknowledgments

I'd like to thank Dr Jenkins for patiently helping me to identify flaws in both my system and my explanation, Drs Doud and Nielsen for helping with quality and context, Sam Dittmer for thinking like a hacker, Michael Barrus for finding hidden graph theory, and Erin Durham for proofreading and the promotion of sanity.

Contents

Chapter 1. Public-Key Cryptography

Symmetric-key cryptography is intuitive and often effective, but it requires that any two parties wishing to communicate securely somehow exchange a secret key. Public-key cryptography makes it possible for any two parties to set up secure communications using only a public, insecure channel, such as the internet.

1.1 The Need for Public Key

At its core, cryptography is about sending messages securely over insecure channels. Most common channels of communication are inherently insecure. Letters can be opened, phone lines can be tapped, and data packets sent over the internet can be read and/or copied. In all of these cases, an attacker can steal a message en route while ensuring that it still makes it to the intended recipient. With care, an attacker can do this such that neither sender nor receiver knows what happened. This is a problem.

In theory, one solution to this problem would be to create a truly secure channel. For example, two people who wanted to communicate securely could run a cable between their computers and use it exclusively to carry their messages. In most cases, this is impractical or even impossible.

A better solution is for two parties to communicate in their own secret language. Then they can use an insecure channel without worry. Even if an attacker intercepts their messages, he cannot obtain any information if he cannot translate the secret language. In practice, this is achieved by treating messages as numbers (or other mathematical objects) and using function-inverse pairs to 'translate' between normal and encrypted messages.

Suppose $f(x)$ is an invertible function. If two parties both know this function, but nobody else does, then one can take a message m, compute $c = f(m)$, and send c to the other over an insecure channel. The other computes $m = f^{-1}(c)$ to recover the message. To an attacker, c is meaningless.

This is the idea behind symmetric-key cryptography [12, p. 71]. The key in this case is the function $f(x)$, which is known to both parties but secret to all others. It is symmetric in the sense that both parties have the same information and are encrypting and decrypting messages in the same way. Of course, this view is extremely simplistic. In reality, it is difficult to make sure that an attacker cannot work out what $f(x)$ is and compromise the system. Today, however, there are methods of generating such an $f(x)$ — such as the AES [16, p. 102] and Twofish [13] algorithms — which are considered unbreakable.

The problem with symmetric-key cryptography isn't that it is easy to break. It isn't. The problem is getting it set up in the first place. In order to do so, the two parties must share some secret information [15, p. 123]. Unless they can meet in person, they are stuck. The logic is perfectly circular — they can't communicate securely unless without first sharing a key, and they can't share a key without first securing their communications.

Public-key cryptography exists to circumvent this apparent barrier. This is accomplished through the use of two different keys: a public key and a private key. A person with the private key is able to encrypt messages, but does not have the capability to invert the encryption process and decrypt messages. A person with the private key can decrypt any message encrypted with the public key. Additionally, the private key usually gives the information necessary to recover the public key and encrypt messages [15, p. 144].

So an individual will generate a public key-private key pair and publish the public key for all to see while keeping the private key a secret. Then any other individual can use the public key to send an encrypted message to the first individual (who created the keys), knowing that no other can decrypt it. No prior communication is required between the two parties involved.

It is easy to design a function that cannot be inverted, and it is easy to design a function which anyone can invert. It is difficult to give somebody enough information to compute values of an invertible function but not enough to compute inverse values or discover the inverse function.

1.2 Trapdoors, Factorization, and Discrete Logarithms

A trapdoor is a door which sits flush with a wall or floor, having no handle on its outside (the side toward which it opens). Thus it is easy to open from the inside, but difficult or impossible to open from the outside. A trapdoor function $t(x)$ is an invertible function — that is, given an output y, it is possible to find the unique x such that $t(x) = y$ — with the property that it is easy to compute $t(x)$ for any x, but difficult to find the x for a given y [15, p. 145].

A common and important example of such a function is $f(p, q) = pq = n$, which takes as its input a pair of primes p and q (order not important) and returns their product n. Given p and q, it is trivial to compute n. Given n, however, it is very difficult to compute p and q even though we know that the pair exists and is unique [15, p. 145].

When we say that factoring is difficult, this is not a statement of fact regarding some sort of inherent difficulty to the problem. Rather, it means that there is no known method for reliably computing p and q in a reasonable amount of time. If someone were to discover a fast algorithm which consistently factored semiprimes (products of two prime numbers), this would no longer be considered a trapdoor function, and the problem of factorization would no longer be called difficult.

Another important trapdoor involves exponentiation in finite groups — for example, the multiplicative group of integers modulo some integer n. Let $g_a(x) = a^x$ be a function that takes a fixed group element a and returns its xth power. This is generally easy to compute, especially with optimizations like binary exponentiation. Then $g^{-1}(x) = \log_a(x)$ is called a discrete logarithm. Unlike logarithms of real numbers, discrete logarithms are infeasible to compute in many groups [17, p. 201]. In fact, this problem is closely related to that of factorization, and algorithms for one tend to have analogues for the other. There are other trapdoor functions currently being studied. Of particular interest are some involving lattice reduction and coding theory, but so far no cryptosystem using these has been widely adopted.

Public-Key cryptosystems are built around trapdoor functions. The trapdoor sits between the public and private keys in the sense that it is easy to derive the public key from the private key, but computationally infeasible to derive the private key from the public key. We will give several important examples of this which are in use today.

1.3 RSA, ElGamal, and Elliptic Curves

No discussion of modern public-key cryptography can be complete without an explanation of the RSA algorithm of Rivest, Shamir, and Adelman. It is both one of the oldest notable examples of a public-key cryptosystem (first published in 1977) and one of the most important and common systems in use today [15, p. 144].

To create an RSA key pair, a user first chooses two large primes (hundreds of digits long) p and q and computes $n = pq$. He then chooses an integer e and computes its inverse d modulo $\varphi(n) = (p-1)(q-1)$. The user publishes e and n as the public key and keeps p, q, and d secret. To send a message m to this user, one simply computes $c = m^e \bmod n$ and sends c. He then decrypts by computing $m = c^d \bmod n$ [19, p. 89]. It is hard to imagine a cryptosystem more elegant or simple in its mathematical description.

The security of RSA lies in the fact that an attacker would need to compute d in order to decrypt messages, but this requires knowledge of $\varphi(n)$ or, equivalently, p and q. It is generally believed that the best way to attack RSA is to factor n [16, p. 182]. Successful factorization makes the rest of the attack trivial, but current mathematical methods and computer technology cannot factor n as long as p and q are chosen sufficiently large. There are a few caveats — there are special-case factoring algorithms that can succeed if p and/or q have certain special properties [4, p. 412] — but these details are not a major concern. The RSA system should be secure as long as factoring is hard and no better method of attack is doscovered. Consequently, integer factorization is currently a major area of research in both academia and industry.

The discrete logarithm problem gives rise to a variety of public-key cryptosystems, most notably the ElGamal system of Taher Elgamal [6] and a family of systems based in the theory of elliptic curves [7]. Whereas RSA essentially forces an attacker to factor the product of two large primes, these systems force an attacker to compute discrete logarithms over groups for which this is difficult. Some of these systems, including ElGamal, introduce an aspect of randomness into the encryption process [17, p. 212]. This makes it possible for one m to be be encrypted as many different c, any of which would be decrypted correctly with the private key. In practice, this can be an important advantage.

These systems are not without drawbacks. Otherwise, there would be little sense in coming up with a new public-key cryptosystem. The most important drawback is that while all are considered reasonably secure now, that security might crumble with the advent of quantum computing. In 1994, Peter Shor described an algorithm which quickly factors large integers on a quantum computer [11, p. 63] using discrete Fourtier transforms [1, p. 25]. A variant of this algorithm efficiently computes discrete logarithms as well [14]. However, so far quantum computers have not been built with enough size or speed for these methods to be implemented on a practical scale. If the technology improves, secure communications everywhere could be totally compromised and the aforementioned cryptosystems rendered useless. The principal value of this paper lies in the presentation of a public-key cryptosystem which would be secure not only against attacks with classical computers, but also against Shor's algorithm and other quantum computing methods.

Another drawback of lesser but real importance is that public-key methods tend to be slower than their symmetric counterparts because they involve number-theoretic operations (such as exponentiation in groups) for which computers are not optimized. This is in contrast to, for example, the AES (American Encryption Standard) algorithm, which uses only the simplest operations (mainly bitwise additions and table lookups) and so is very fast [17, p. 151]. Other symmetric-key methods are similarly fast.

It would be ideal to have a system with the unique advantages of the public-key setup and the speed of comparably secure symmetric methods, and the system to be presented is designed to be just that.

1.4 Digital Signatures

In addition to removing the need to exchange a key securely, public-key systems provide a solution to the important problem of verifying a message's source — that is, confirming that the sender of a message is indeed who he claims to be [16, p. 274]. The ability to communicate privately is of little value if the identity of the other party is uncertain, and credentials like email addresses are surprisingly easy to fake.

The verification is accomplished essentially by using a cryptosystem in reverse. Suppose two users, Alice and Bob (who have been suspiciously absent from the explanation thus far), each have their own implementation of a public-key cryptosystem, and Bob wants to send a message to Alice such that can be sure that only he could have sent it. To do this, Bob takes his message and uses his private key to 'decrypt' it as though it were an incoming encrypted message. He then encrypts the resulting message with Alice's public key and sends it off. Upon receipt, Alice decrypts the message with her private key just as she would any other incoming message. She then uses Bob's public key to 'encrypt' the resulting message, undoing Bob's initial decryption and yielding the original message. If Bob's public key undoes an operation, it must be the operation of Bob's private key, which only Bob has, so the message must be from Bob [15, p. 213]. This underscores the importance of keeping a private key private. If an attacker can gain access to Bob's private key without Bob's knowledge, he may be able to impersonate Bob with this method.

1.5 Symmetric Key Exchange

Because of the speed discrepancy between public-key and symmetric encryption, it is common in practice to implement a hybrid cryptosystem that takes advantage of the strengths of each. In this case, two users create public-key systems and use them only to securely exchange the information to set up a symmetric system, which they then use for all subsequent communication [17, p. 259].

Chapter 2. The Encryption and Decryption Functions

An instance of the system comprises a bijective function on the message space. We describe this function and its inverse, the space upon which they act, and the keys that facilitate their computation.

2.1 Bits, Words, and Blocks

The role of cryptography today is to secure the transmission of information stored on computers as strings of bits. It is natural, therefore, that a modern cryptosystem should be designed to work with data in this form. With that in mind, we define a few useful terms.

First, let α be a small positive integer. A 'word' is a string of α bits. For example, if $\alpha = 4$, then 1101 and 0110 are both valid words. There are 2^α possible words, so for convenience define $n = 2^\alpha$. Let m be another positive integer. A 'block' is a list of m words. An instance of the system will send a block of plaintext to a block of ciphertext and vice-versa, and so may be thought of as a bijection on the set of blocks.

Let W be the set of words. It is worth noting that W is just that — a set. It will take on different algebraic structures at different stages and it will be important to keep track of how elements translate. Good values for α and m will be discussed later, when the proper context has been established. To give an idea of scale, however, α will likely be around 10, and m somewhere around 100.

2.2 Permutations

The symmetric group S_n is usually described as the group of permutations on n symbols [8, p. 46]. Equivalently, we describe it as the group of bijective functions on the aforementioned set W of n possible words, with composition of functions as the group operation. We're going to use lots of functions from this group.

Let $S_n^m = \prod_{i=1}^m S_n$ be the group (under composition) of functions on the set of blocks which operate on each word separately (so this is a small subgroup of S_{nm}, which consists of all bijections on the block as a whole). A function $f \in S_n^m$ is written as the tuple $(f_1, f_2, \ldots, f_m)$. If we write a block P as $[p_1, p_2, \ldots, p_m]$, then $f(P) = [f_1(p_1), f_2(p_2), \ldots, f_m(p_m)]$. Such functions will form a sort of 'buffer' to keep attackers from gaining information about the encryption operation.

2.3 Finite Field Arithmetic

Let $q(x) \in F_2[x]$ be irreducible of degree α and let $F_n = F_2[x]/(q(x))$ be the finite field of $n = 2^\alpha$ elements where $F_2 = \mathbb{Z}/2\mathbb{Z}$ [8, p. 278]. The elements of F_n are polynomials in x with coefficients in F_2 and of degree at most $\alpha - 1$. We will identify the elements of F_n with the words of length α in an obvious way, by letting the coefficient of x^i correspond to the ith digit (from the right, counting from zero) of the word. For example, if $\alpha = 4$, we identify $x^3 + x^2 + 1$ with 1101 and x with 0010. We then can speak of adding and multiplying words by carrying out the corresponding operations in F_n [15, p. 250].

Addition in particular is very easy, as it amounts to performing a binary XOR on the words [17, p. 40]. So $1100 + 0111 = 1011$ and $1011 + 1010 = 0001$. Conveniently, F_n has characteristic 2, so addition and subtraction are the same operation. Hence we can code arithmetic only in terms of the former.

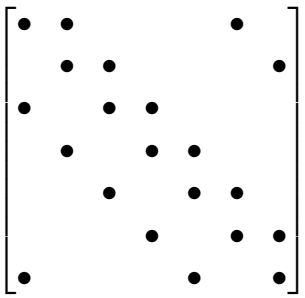
Multiplication is not trivial (especially by hand), but can be implemented efficiently. To naively compute the product of two words, we would compute the product of two polynomials and then reduce modulo $q(x)$. For example, let $\alpha = 4$ and $q(x) = x^4 + x + 1$. Then $0101 \cdot 1011 = (x^2+1) \cdot (x^3+x+1) = x^5+x^2+x+1 = 1 = 0001$, with the last step using the fact that $x^5 \equiv x(x^4) \equiv x(x+1) \equiv x^2 + x \bmod x^4 + x + 1$. Division is possible in the usual 'multiplication by the inverse' sense, so in order to perform division one would have to know these inverses, which are easy to compute by exponentiation using Fermat's Little Theorem.

In the case where α is very small, it might be advantageous to pre-compute a multiplication table. However, this table quickly becomes very large. A log table expressing the various field elements as powers of a generator (x is an obvious choice) takes up much less space and turns multiplication into addition and subtraction of small integers. Then our previous example becomes $0101 \cdot 1011 = x^8 \cdot x^7 = x^{15} = 1 = 0001$.

2.4 A Matrix Twist

Let $GL_m(F_n)$ be the group (under multiplication) of $m \times m$ invertible matrices over F_n [5, p. 34]. The most important element of the system will be a matrix $T \in GL_m(F_n)$. In addition to its invertibility, T needs to have a special shape.

First, T needs to have exactly three nonzero entries in each row and in each column. The importance of this condition will be explained later, in terms of a graph closely related to T. Given the expected size of m, this condition makes T very sparse. Here is an example of one possible shape for T in the case where $m = 7$, in which the bullets represent nonzero entries:



In order to use T, we interpret a block P as a column vector over F_n. Then we can multiply this block by T on the left to get a new block. This operation is essential because it 'entangles' a block at the word level, in the sense that the words in the new block are functions of several words from the old block. This is in contrast to the action of functions from S_n^m, which operate on words individually.

2.5 The Private Key

We now have all of the pieces necessary to describe the actions of the encryption and decryption functions. Let $F, G \in S_n^m$ and T be as above. Then define

$$E(P) = G(T \cdot F(P)) \text{ and } D(C) = F^{-1}(T^{-1} \cdot G^{-1}(C))$$

Here P is a block of plaintext and C is its corresponding block of ciphertext. They can be obtained from one another by E and D, which are easily seen to be inverses.

The private key can consist simply of F, G, and T. The functions F and G may be stored rather naively as lists of outputs which are treated as lookup tables. For example, in the case $\alpha = 2$, $m = 3$ we might write f_1 as [01,11,10,00], meaning $f_1(00) = 01$, $f_1(01) = 11$, $f_1(10) = 10$, and $f_1(11) = 00$. Writing f_2 and f_3 likewise, we can store F as a 3×4 array of 2-bit words.

This method may not be strictly optimal in terms of information density when α is larger, as it takes $n\alpha = \log_2(n^n)$ bits to store each S_n function when one could theoretically use only $\log_2(n!)$ bits for each, but the added complexity would almost certainly not be worth the space saved. The first part of the private key consists of F^{-1} and G^{-1} stored as described here. Because F and F^{-1} take up the same amount of space, it is better to store F^{-1} than to store F and waste time inverting it before each use.

It will, on the other hand, be worth storing T instead of T^{-1} for two reasons. First, T can be stored in very little space because of its sparsity, which T^{-1} cannot be expected to share. Second, the method described here will come in handy later when publishing the public key.

We store T using two $m \times 3$ arrays called S and V. S stores the 'shape' of T by listing in ascending order the three columns in which each row has its nonzero entries. In the case of the diagram on the previous section, S would be $[[1, 2, 6], [2, 3, 7], \ldots, [1, 5, 7]]$. Technically S could be stored a bit more efficiently (for example, the last row could be inferred from the previous), but not by a meaningful amount.

The nice, banded shape of the example might invite an even more streamlined representation, but in general this will not be the case. The array V then stores the actual nonzero entries of each row, in order from left to right so as to match the way columns are listed in S.

T is easy to invert using conventional methods. In particular, inverting a matrix over a finite field is very nice computationally as stability and precision are not concerns (as they are when working with, for example, floating-point reals). In most cases, inverting T before beginning decryption will not add meaningful time to the process. So the second part of the private key consists of T stored as described.

Of course, one could publish F, G, and T in the same form and allow others to encrypt, but this would also make public enough information to easily decrypt. In order to assemble a public key that does the former but not the latter, new pieces are needed.

2.6 Affine Interjection

Let P and C be a plaintext-ciphertext pair again. It is easy to see that an individual word c_i of C is a function of three words of P - the words corresponding to the three columns in which row i of T has nonzero entries, or the three words indicated by row i of S. That relationship looks like this:

$$c_i = g_i\left(t_{s_{i,1},i}f_{s_{i,1}}(p_{s_{i,1}}) + t_{s_{i,2},i}f_{s_{i,2}}(p_{s_{i,2}}) + t_{s_{i,3},i}f_{s_{i,3}}(p_{s_{i,3}})\right)$$

For greater generality, we define the function

$$e_i(x,y,z) = g_i\left(t_{s_{i,1},i}f_{s_{i,1}}(x) + t_{s_{i,2},i}f_{s_{i,2}}(y) + t_{s_{i,3},i}f_{s_{i,3}}(z)\right)$$

and note that $E(P)$ can be broken down into the sequence of e_i in order to compute the words of C individually. It's a bit more complicated conceptually, but computationally equivalent to what was described before - just with the steps re-ordered.

Still in the spirit of making changes with no net effect, we define just a few more objects. Let $a_i, b_{i,1}, b_{i,2}, b_{i,3} \in F_n$ with a_i nonzero. Let $b_i = b_{i,1} + b_{i,2} + b_{i,3}$. Then we define $g'_i, f'_{i,1}, f'_{i,2}, f'_{i,3} \in S_n$ by

$$g'_i(x) = g_i((x - b_i)/a_i) \text{ and } f'_{i,j}(x) = a_i t_{s_{i,j},i} f_{s_{i,j}}(x) + b_{i,j}$$

for $1 \leq j \leq 3$. Then it is clear that

$$e_i(x, y, z) = g'_i \left(f'_{i,1}(x) + f'_{i,2}(y) + f'_{i,2}(z) \right)$$

even though this composition involves four different functions (which are still in S_n) than those in the definition of e_i above. It is this fact that will allow the existence of a public key that facilitates encryption but not decryption.

2.7 The Public Key

Using the same lookup-list method of representing S_n functions as before, we can put these newly minted g'_i and $f'_{i,j}$ into convenient arrays. Specifically, we define arrays E_i for $1 \leq i \leq m$ such that the rows of E_i are, in order from top to bottom: g'_i, $f'_{i,1}$, $f'_{i,2}$, $f'_{i,3}$. These m arrays are all $4 \times n$ and their entries are α-bit words. Of course, in order to encrypt, one must know which plaintext words to plug into these functions, so the final array included in the public key is S - the same S from the private key that locates the nonzero entries in T.

For any f_i in the private key, there are three $f'_{i,j}$ in the public key which differ from it (and from each other) only by composition with an affine function. One could cut the size of the key roughly in half by only giving only one of the three related $f'_{i,j}$ and then providing the necessary constants to derive the other two therefrom. For the sake of simplicity, we won't worry about this optimization here, but it might be worthwhile in practice.

2.8 Implementation

In practice, it is important that a key pair for any system be generated with good pseudo-random methods [17, p. 41]. It is trivial to use a good random number generator iteratively to make the many permutations in the private key, and to generate all of the a_i and $b_{i,j}$ used in making the public key. Filling in the nonzero elements of T randomly could result in a singular matrix, but in this case a random (nonzero) change to any element should fix that.

Choosing a shape for T is not at all a random process, as this shape must give certain specific graph-theoretic properties to be discussed later. However, because the shape is public, there is no reason to 'generate' one at all. In fact, it is likely that if this system were widely implemented that all users would simply select from a handful of particularly good shapes, and this would not present a problem.

Chapter 3. Reverse Engineering

The best thing one can hope to do when attacking a public-key cryptosystem is deduce the private key from available information, so it is paramount that this be infeasible.

3.1 The Symmetric Buffer

The most important part of the system is the matrix T because it creates asymmetry — the defining characteristic of a public key system. Because T has its particular sparse shape, but T^{-1} does not, it is possible to give all of the information needed to encrypt a message without giving the information needed to decrypt one. By itself, however, the action of T is just a linear transformation, and a particularly nice one at that. The reason F and G exist is to prevent an attacker from gaining information about T from either side without sacrificing the speed and simplicity of the system.

Of course, when it comes to information about a linear transformation, the sparsity of T together with the shape information available from the public key would seem to be a gold mine. However, m and n don't need to be very large for the set of possible T with a given shape to be enormous. Since there are $3m$ entries in T, there are $(n-1)^{3m}$ ways to fill them in with nonzero elements of F_n. It is possible that some of the $(n-1)^{3m}$ possible T will fail to be invertible. It seems reasonable to assume (though it might be hard to prove) that the determinants of the possible T should be distributed equally, or nearly so, in F_n. As every element of F_n is a unit, this would make us expect $(n-1)^{3m+1}/n$ valid possible T for a given shape.

The purpose of the F and G 'buffers' is then not to obscure in any way the shape of the transformation, but rather to make it infeasible to determine the values in V. We shall discuss how this is accomplished, but first we must show that an attacker needs to discover the values in V.

3.2 Equivalent Keys

This system is unusual in that a given key - public or private - does not correspond to a unique partner. The method of constructing a public key makes this obvious, with the random selection of a_i and $b_{i,j}$ yielding some $n^{3m}(n-1)^m$ options that all encrypt identically. Given one public key, it would be easy to derive many others, though there would be neither enough time nor enough memory to compute any significant proportion. If one could take a given public key compute all of the equivalent public keys, one of them would consist of all of the original g_i and f_i (three times each) from the private key.

Likewise, two private keys can can decrypt identically. Multiplying T on the left or right by an invertible diagonal matrix and making commensurate changes in the f_i or g_i respectively would yield an equivalent private key. This yields $(n-1)^{2m}$ identical private keys, and these could in turn give rise to an even greater set of public keys equivalent to those described in the previous paragraph.

First, it's important to note that while these classes of equivalent keys are quite large in one sense, they are very small relative to the set of possible keys. Given a particular matrix shape, one could generate $(n!)^{2m}(n-1)^{3m+1}n^{-1}$ different private keys, and dividing into equivalence classes of the aforementioned size barely makes a dent in this number. So we won't run out of non-equivalent keys, nor need we worry about the probability that two keys generated independently will be equivalent (assuming good pseudorandom number generation, of course).

Second, the issue of equivalent keys is not necessarily cause for alarm. It just requires a paradigm shift when considering possible attacks. In this chapter, that means analyzing the difficulty of finding a private key equivalent to the desired one. It's not a tremendous shift, but it merits explanation. Since we will be thinking in terms of equivalence classes of keys, it is natural to want canonical representatives for these classes. This simplifies the discussion and allows us to talk about these representatives as though they were the classes, as we do in modular arithmetic. The best representative will depend on the attack being studied.

3.3 Affine Pass-Through

Another issue in showing that an attacker must get through the buffers, so to speak, is that linear operators like matrices and our affine functions tend to play very nicely together, interacting in simple and predictable ways [9, p. 175]. In particular, scalar multiplication passes right through matrix multiplication. Since the only thing separating the private and public keys is a handful of affine functions, this sounds problematic.

If we want a working private key, we need an F, a G, and a T. In a sense, we have the latter two. We can use the g'_i as the g_i and build a T from the given shape with ones for all the nonzero entries. There is no expectation that these functions would initially help in decryption - and this T might not even be invertible - but they are a logical starting point. The first real hurdle to overcome, though, is choosing some f_i.

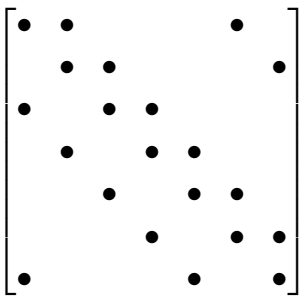
The public key gives three permutations $f'_{i,j}$ for each index i with relations of the form $f'_{i,j}(x) = f_{i,k}(rx + s)$ for some $r \in F_n^\times$ and $s \in F_n$. Note that here it seems prudent to use r and s here instead of a and b to avoid confusion with the a_i and $b_{i,j}$ used to create the $f'_{i,j}$. Given $f_{i,j}$ and $f_{i,k}$, it should be trivial to work out the r and s that identify their relation by matching two outputs from each. Further, it would be easy to take any of the three $f'_{i,j}$ and generate a list of $n^2 - n$ possible f_i. All three $f'_{i,j}$ would of course appear in this list.

Out of the $(n^2 - n)^m$ possible F one could choose by selecting an f_i from each of these lists, it is likely that none would work with the naive choices for G and T. Because the only equivalent private keys are the ones described in the previous section, just using the g'_i won't work unless $b_i = 0$ for every $1 \leq i \leq m$. In other words, even though it is easy to generate the $n^2 - n$ possibilities for each f_i and g_i, being able to generate an equivalent private key to the original is equivalent to knowing when the constant in the affine transformation is zero, because while multiplication by constants can 'pass through' the matrix multiplication under some circumstances, addition just cannot. There's just no way to look at a permutation individually and say whether this has happened. Plus, all of this assumes guessing the entries of T correctly, which is certainly unlikely.

Clearly there are too many things to guess simultaneously before one can check whether those guesses are good by seeing if the resulting potential private key works as a whole. An attacker needs a way to guess some parts of the private key and then verify them before moving on to the next. In order to show that this is difficult, we're going to need to take a detour through a branch of mathematics not usually associated with cryptography, especially of the public-key variety.

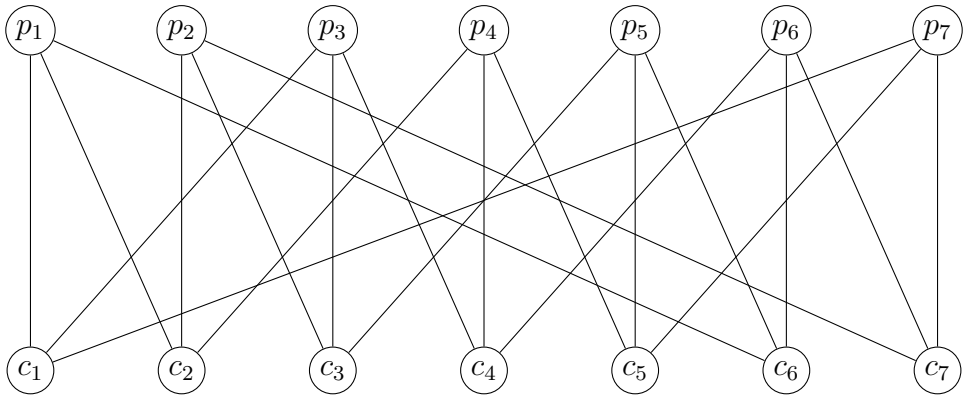
3.4 Girth

In chapter one, we talked about shapes for T, stating that it needed to have three nonzero entries in each row and column without considering why. Now we have the context to give motivation. We return to the example shape for the $m = 7$ case for convenience.



The connections between plaintext words and ciphertext words that we see in this diagram are more naturally understood in the context of graph theory. As the rows of T correspond to words of the plaintext and the columns to words of the ciphertext, having a nonzero entry in the (i, j) position indicates a connection between p_i and c_j. So we draw a graph in which the vertices represent words and the edges represent these connections. It will be an undirected graph, which might be counterintuitive as the encryption operation has a definite direction. This is because an attacker can 'travel' along edges in both directions in a manner to be discussed later.

In the case of the above example, one might draw this graph as follows:



Drawing the graph in this way makes a few features obvious. First, it is necessarily a balanced bipartite [18, p. 3] graph on $2m$ vertices, as both the plaintext and ciphertext blocks consist of exactly m words and there cannot be connection between words in the same block. Second, it is obvious from the way T was defined that this is a cubic (or 3-regular) graph [10, p. 15].

This example is unusual in that its structure is simple and easy to see — even pretty. For larger m, things will be messier. More importantly, this graph is unusual in that it exemplifies a trait that will prove crucial to the security of the system: high girth. The girth of a graph is the length of its shortest cycle. It will be useful to think of girth in terms of a 'cycle game' in which a player starts at a vertex and follows edges around the graph, attempting to return to the starting vertex in as few moves as possible without using any edge twice. Note that because our graphs are bipartite, their girth will always be even.

The above graph is known as the Heawood graph and has a girth of 6 [3]. In fact, it is a 6-cage — it has the least vertices of any cubic graph with girth 6. The converse is true as well - 6 is the highest possible girth for a cubic graph on 14 vertices. It is known that a graph of even girth g must have at least $2^{1+g/2} - 2$ vertices [2]. So the shape shown above is an ideal choice for T when $m = 7$. Permuting the rows and/or columns of T will not change the graph at all, so in this sense there are many equally good shapes.

If one were going to make an instance of the system with $m = 7$, it would be foolish to use any shape but these. Any other would have less-than-optimal girth, making the system less secure. There is no value in using a different graph than others because it is published in the public key (in the form of the array S).

3.5 The Cycle Game

Let's continue with our example. In it, c_1 depends on p_1, p_3, and p_7. Suppose an attacker has guesses for f_1, f_3, f_7, g_1, and all the relevant entries of T. By plugging in several triples of values for p_1, p_3, and p_7, we will have a high chance of quickly identifying our set of guesses as defective when some triples don't give the right c_1 as output. While this is still crude from a strategic viewpoint, it's much less expensive than guessing a whole private key and testing it for equivalence to the desired one. On the other hand, having guesses that pass this test is no achievement in itself. A trivial way to pass is to use g'_7, its three corresponding $f'_{i,j}$, and ones for the T values. An attacker can generate every passing set by picking random constants in a method almost identical to the one used in making the public key, with one small addition.

In order to generate such a set, an attacker starts with the g'_i and its $f'_{i,j}$. He then randomly generates $r, t_1, t_2, t_3 \in F_n^\times$ and $s_1, s_2, s_3 \in F_n$. and defines $s = s_1 + s_2 + s_3$. He then makes the following guesses:

$$g_i(x) = g'_i((x - s)/r) \text{ and } f_j(x) = (r f'_{i,j}(x) + s_j)/t_j$$

(where j takes the values of the indices of the three plaintext words that determine c_i) for the true private key functions. The difficulty lies in trying to extend this particular set of guesses all the way to a full private key without it failing. The difficulty of doing so is a function of the girth of the graph just described.

Suppose we have a set of guesses as above. In particular, look at the f_1 from that set. Since f_1 is also used in the computation of c_6, it needs to pass a test there as well. So we need to come up with guesses for f_5, f_6, g_6, and a few more entries of T that work with our f_1. This means repeating the process just outlined with the r, s_j, and t_j but with the constraint that the constants chosen yield the same guess for f_1 that we have already made. This will uniquely determine r and one each of the s_j and t_j (the s_j and t_j corresponding to f_1). The rest of the s_j and t_j can be chosen randomly.

We can try to continue this process with c_7, but trouble will almost certainly arise. In this case, focusing on c_7 would have us look at f_2, f_6, and f_7. We already have guesses for two of these, and the new guesses we make must agree with both. However, it is highly likely that our f_6 and our f_7 will put contradictory constraints on the choices of constants. If so, it's the end of the road.

Recall from the section on equivalent keys that some multiplication by constants can leave a private key intact, but addition of constants cannot because they can't pull through the matrix action. So if at any stage in this guessing game the s constants failed to cancel out the b constants added when the public key was made, we are doomed to fail. Yet this can only be seen upon completing a cycle of the graph, at which point the contradiction was apparent. This is the importance of girth. The more steps an attacker must perform on the way to this contradiction, the more unlikely his success.

In fact, by keeping track of random guesses, we can see that the difficulty of successfully closing the cycle increases exponentially with the girth g of our graph. If we only consider the possibility of not cancelling out the added constants (and there are other ways to lose this game, as T can be picky), an attacker must guess three correct s constants on the initial step, which will happen 1 in n^3 times. Then with each successive step he must guess an additional two (assuming $g > 4$, which prevents two c_i from having multiple p_i in common). Then $g/2$ steps to a cycle gives a 1 in n^{3+g} (or 1 in $2^{\alpha(3+g)}$) chance of success. As just stated, this considers only one reason for failure and so constitutes a weak lower bound on difficulty.

This finally answers the critical question left unanswered in chapter two: how does one choose a good shape for T? It is smartest to choose α and an even girth g simultaneously so as to make the cycle game sufficiently difficult as per the formula just given. Then choose a balanced bipartite cubic graph of girth g either by generating one or (more likely) by using a well-known example. In particular, it is natural to look at the known cages, as these will result in the smallest m for a particular girth, which in turn will keep both key size and runtime down. Then one can use any shape for T which is equivalent to the selected graph (of the many given by permuting rows and columns).

3.6 Plaintext-Ciphertext Pairs

In symmetric encryption, giving an attacker access to even a small number of pairs of corresponding plaintexts and ciphertexts usually spells doom. Of course, in the public-key case, an attacker has immediate access to every such pair. It is important to show that one cannot generate a set of such which reveals helpful information about the private key. We are not considering cases in which an attacker chooses a ciphertext and somehow (through either implementation or user error) obtains the corresponding plaintext. This situation will be treated in the following chapter. Here an attacker is alone with the public key, encrypting plaintexts and looking for information in the output.

Preventing such an attack is another crucial function of the permutation buffers. Essentially, there are no 'special' plaintext-ciphertext pairs for this system because both sides are randomized. There are many special vectors which can give one information about matrix multiplication, such as the zero vector and the elementary basis vectors, and the eigenvectors of a matrix. However, not only can an attacker not intentionally generate a plaintext which will become such a vector after passing through F, he cannot even tell if a given plaintext will become such a vector. This is why the f_i and g_i were chosen from S_n — for maximum randomness — though in chapter six we will consider choosing them from a smaller group.

It is difficult to imagine even a large list of plaintext-ciphertext pairs giving any more information about the private key than is readily apparent just from looking at the functions in the public key. This is true to some extent for any public-key system, but particularly so in this case as there is so much randomness obscuring the structure of the system.

3.7 Partial Private Keys

Finally, we ask what an attacker could do if he could obtain a partial private key. It seems unlikely that many situations would arise in which only some of a key was leaked, but as this happened just recently to servers around the world, it merits consideration. To make a long story short, several types of partial private keys would facilitate effective methods of recovering the rest. As any reasonable person would conclude regardless of this fact, any leak - partial or complete - is more than sufficient reason to generate a new key pair.

In particular, if an attacker had several f_i and/or g_i, it could render the cycle game trivial by reducing the factor of increase in difficulty per step from n^2 to n or even 1 if the relevant functions to be guessed were available. Further, once one cycle is completed, it is possible to complete other cycles that share several edges with it on the graph much more quickly. Fortunately, the key space is so large that one user's leak can be safely expected to leave all others unaffected if generation is at all random.

Chapter 4. Cryptanalysis

If the private key is safe, the next priority is naturally to assure that there is no reasonable algorithm for working out individual plaintexts.

4.1 Differentials

One intuitive but powerful technique in the analysis of a public-key system is to encrypt closely-related plaintexts and look at the differences in the corresponding plaintexts [17, p. 118]. In our case, it might be natural to choose two plaintexts which differ in a single word. In this case, the three ciphertext words depending on that word would change while the others remained the same. Turning this around, an attacker looking at ciphertexts with many similar words can deduce that their plaintexts may differ in only a few words, and he can determine which words those are.

In a naive implementation of the system, this might indeed be a problem, especially if the plaintext being encrypted had exploitable structure that lined up with the division into words. For example, if $\alpha = 8$ and the data to be encrypted consisted of English text encoded with UTF-8, it might be easy to tell if two blocks of the text differed in a small number of characters. Other types of data might be similarly exploited by an attacker aware of the context of the message.

While the system as described here does not address this concern directly, it it not hard to sort out with existing cryptographic methods. So far we have treated the system as an electronic code book-type block cipher as it is convenient for most theoretic discussions, but in practice it would make more sense to use a mode of operation such as cipher block chaining (with a random initialization vector) or counter mode. This would prevent ciphertext blocks from revealing if their plaintext blocks were similar or even identical [17, p. 131].

4.2 Distinguishing

Another classic problem in public-key cryptanalysis is that of 'distinguishing.' Essentially, an attacker wants to be able to look at a ciphertext C_1 and several plaintexts $P_1, P_2, \ldots$ and determine quickly which of these plaintexts was encrypted to give C_1. With a deterministic system, this problem is trivial to solve. One simply needs to encrypt all of the proposed plaintexts and look for a match. This system is not only deterministic, it's bijective. Every possible block of ciphertext corresponds to a unique block of plaintext, so distinguishing is as easy as it could ever be.

This could be changed through the use of random padding, at the expense of efficiency [16, p. 128]. It would be silly to do so, however, without establishing the existence of a problem. Clearly if an attacker knows that a message comes from a small set, he can use this fact to find out which it is. So if you're just going to send the message "Yes" or "No," and an attacker somehow knows this, you should probably fill the rest of the message with random bits rather than all zeroes. Otherwise, a good mode of operation should handle this problem as well by making it impossibile to generate a set of possible inputs for a given block without access to insider information.

4.3 Plaintext Leaks

If there is insider information available in the form of a partial or complete plaintext, then there may indeed be a problem. Of course, this would be equally true for any cryptosystem, but it is reasonable to examine the nature of that problem in this specific case.

In the case of a partial leak, the cycle game comes back into play. If an attacker has two of the plaintext words that determine a certain word of the ciphertext, it is trivial for him to determine the third. With a relatively small number of the words, one can work out the rest with a sort of domino effect. As m becomes large, it is reasonable to expect the number of plain words necessary to grow logarithmically like maximum girth.

In our by-now-beloved example, say an attacker has p_1, p_6, and p_7 in addition to the ciphertext. Using p_6, p_7, and c_7, he obtains p_2. Using p_1, p_2, and c_2, he then obtains p_4. There are then multiple options for working out p_3 and p_5. This hints at an interesting problem in graph theory that might be studied in its own right: given one of these graphs and all of the c_i, what is the smallest set of p_i that could be used to work out all the others in this manner?

So a partial plaintext leak can quickly become a full leak. As stated in the previous chapter, this doesn't allow an attacker to work out the private key and start decrypting other ciphertexts. However, the idea of working out the rest of a partially leaked plaintext is important because it is the basis of the most important attack against the system. Fortunately, it's also the one the system is carefully designed to frustrate through graph theory.

4.4 Guess and Check

A discussion of brute-force attacks has been suspiciously absent so far. It's not just because the key and message spaces are so large. Just as an attacker can (and should) play the cycle game as a much more efficient (if still untenable) alternative to true brute force, he can use short graph cycles to work out a plaintext more efficiently. It just needs to be shown that this process involves similar road blocks.

Instead of supposing that an attacker has obtained a few plaintext words corresponding to a known ciphertext in some sort of leak, let him guess those values. Specifically, say an attacker guesses a triple of plaintext words that give one of the ciphertext words, as one can easily do. Then he can try to expand this to a full plaintext, checking along the way for contradictions. If there is going to be a contradiction, naturally he wants to find it as quickly as possible, so he will follow the shortest available graph cycle back to one of his original guessed words. If he's off track, his more recent guesses will imply that this word is something other than he originally guessed it to be, showing that the original guess was bad. Then the process has to begin again.

For a given ciphertext word, there are n^2 triples of plaintext words which give it correctly. Specifically, choosing values for any two of the determining words will fix a unique third. From there, an attacker can take any of the three words in the triple, choose another ciphertext word it affects, and look at its triple. He's already got one value, so he chooses one of the others from the n possibilities and the third is determined. This process continues until a ciphertext word other than the ones he has used is fully determined by three of the plaintext words he has chosen along the way. If that ciphertext word is determined to be something other than the one in the actual ciphertext, he must begin again. This cannot happen until he has completed a cycle on the graph, so the attacker has made $1 + g/2$ guesses, each out of n possibilities. The chance of completing a cycle successfully with this method is then 1 in $n^{1+g/2}$, and even then the job of working out a single plaintext block is not done, though the hardest part is over.

4.5 Side Channels

Side channel attacks are fascinating because they don't seek to exploit mathematical structure or software as much as the actual hardware used to run the software and transmit messages. We should show that methods that have worked on other public-key systems are of no use here.

In this system, the time taken for any encryption (respectively, decryption) operation should be identical regardless of the particular message and key. There will be the same number of additions and multiplications in every encryption (respectively, decryption) operation. The additions should all take the same time because they are really bitwise xor operations. In a log table implementation, multiplications become additions of small integers. For these reasons, the electrical power used and other mechanical metrics should give no information to an attacker, even if he is the one choosing the messages. This may not be true for all hardware, however, so it is important to know if a method of addition operates in constant time.

This complete uniformity, or lack of structure in the message and key spaces, makes all sorts of clever cryptanalysis difficult. There aren't patterns to exploit other than the ones the system wears on its sleeve, and we have discussed the exploitation of those. Unless we consider genuine data leaks to be side channel attacks in the loosest sense, such methods shouldn't be relevant.

Chapter 5. Practical Considerations

A secure system is only valuable if it can be implemented with acceptable ease and efficiency in a full range of applications.

5.1 Temporal and Spatial Complexity

It is easy to say exactly which operations (and how many of each) will be necessary in actually computing with this system. To do this for encryption, we just look at the e_i as defined earlier. Each one involves four lookups from lists of length n and two additions in F_n (which are just binary xor operations). We need to compute m of these for a grand total of just $4m$ lookups and $2m$ additions.

Decryption is not quite as fast. Hitting the ciphertext with G^{-1} and F^{-1} will take a grand total of $2m$ lookups. In between those two steps, we must multiply a vector from F_n^m by a matrix from $GL_m(F_n)$ (with no particular sparsity or other special properties) on the left. Doing this naively involves m^2 multiplications (using smart methods like the log table described in chapter one) and $m^2 - m$ additions. There are asymptotically better methods for these mutltiplications, but they are unlikely to be superior for the relatively small m in use here.

Both operations use relatively little computer memory. Decryption can actually be done entirely in place, so in a sense it requires none at all beyond that needed to hold the private key and ciphertext. Encryption only requires holding a function from the public key and a few words at once, so its memory requirement is similarly negligible.

5.2 Key Size

We remember that the public key consists of $4m$ functions from S_n stored as lists and the shape array S. Then it has a size of exactly $4mn\alpha + 3m\beta$ bits if stored as described, where $\beta = \lceil \log_2(m) \rceil$ is the size of indices needed in S.

The public key consists of F, G, V, and T which take up $mn\alpha$, $mn\alpha$, $3m\beta$, and $3m\alpha$ bits respectively. In practical applications, n will be large enough that we can use the rough sizes $4mn\alpha$ for the public key and $2mn\alpha$ for the private key. So if one chose, for example, $\alpha = 8$ (so that words are exactly bytes, for convenience) and $m = 63$ (as the 12-cage has 126 vertices), we'd be looking at keys of about 32 and 64 kilobytes respectively - about an order of magnitude larger than those of some other public key systems, but still tiny for modern technology.

5.3 Scaling

As computers become faster and hold more memory, it could become necessary to increase our parameters so that methods of cryptanalysis discussed remain impossible to complete in a realistic amount of time.

5.4 Portability and Nativeness

This system is designed specifically to work with data types and operations for which all programming languages, operating systems, and hardware types are already optimized on a deep level, so it should work perfectly in any digital setting without special hardware or software accomodations.

5.5 The Human Factor

Of course all of cryptography can be trivialized when implemented poorly by programmers or mishandled by users. These things cannot be prevented mathematically, but designing the system around common data types and simple operations should minimize confusion and error. Beyond that, we can only hope that the hardware being used doesn't already have spy tech built in.

Chapter 6. The Future

Where do we go from here?

6.1 A Smaller Group

The symmetric groups don't discriminate. They allow any invertible function and so are quite large. One could argue that in this application they are much larger than they need to be. Selecting the f_i and g_i from a significantly smaller subgroup of S_n could reduce the sizes of both keys quite a bit. In addition, if this subgroups were chosen for computational properties (for example, one could use a subgroup of polynomial functions), then it might be possible to encrypt and decrypt quickly without having to expand the functions to lookup tables.

The most extreme suggestion with any plausibility would be to select f_i and g_i from the group of affine functions on F_n. Such functions can be represented with only 2α bits each (as opposed to $n\alpha$ bits as before) and computed with minimal cost, perhaps even more quickly than lookups (depending on methods and hardware). Would this compromise security in some way? If so, could a slightly larger or more complex subgroup avoid this downfall while offering similar benefits? It certainly merits investigation, as the gains in lightness and simplicity could be tremendous.

6.2 Efficiency Through Repetition

While the keys for this system aren't terribly large, it would certainly be nice if they could be made smaller without decreasing α or m. In the case of the private key, this could be done rather easily by re-using permutations rather than generating m distinct f_i and g_i. For example, one could choose a very small positive integer k, generate f_i and g_i for $1 \leq i \leq k$ and then say $f_{i+k} = fi$ for the rest of the f_i, and likewise for the g_i. In this case, the key can be stored in a relatively tiny space.

The size of the public key could be similarly reduced by choosing the entries of T and the a_i and $b_{i,j}$ such that the g'_i repeat and there are only a small number of unique $f'_{i,j}$ whose occurences can be described with a small table. As before, it should be reasonable for the public key to be about twice the size of the private key.

Once these smaller keys were expanded for use, encryption and decryption could be carried on exactly as before. It is natural, however, to be concerned that all this repetition could be exploited for an effective attack on the system. In particular, if $k = 1$, one could easily guess the private key by brute force. Is there some way to establish k and rules for selecting the T entries and affine functions such that the keys could be meaningfully reduced without compromising security? In a sufficiently large-scale application, it would merit investigation.

6.3 New Graphs

In the system as described we used cubic balanced bipartite graphs for several reasons. Most importantly, vertices with at most two edges could be trivially bypassed in cryptanalysis, and vertices with more than three edges make high girth harder to achieve, so it seems natural to make the graph cubic. The simple relations between the c_i and the p_i dictates that the graph be balanced bipartite. However, it is conceivable that other (presumably more complex) graphs could make the crucial 'cycle game' more difficult for given n and m. This might require multiple matrix operations — or entirely new operations combining multiple plain words in the computation of individual cipher words — in the encryption process. This could in turn result in spatial gains, but more importantly it could allow for smaller m, thus speeding up the decryption process.

6.4 Rings and Ideals

All of the math here is done over a finite field, but could be generalized without difficulty to any commutative ring. It's tough to imagine going beyond commutative rings because we need to invert matrices — though perhaps this could be done cleverly with LU decompositions - but even the commutative case gives a much wider range of algebraic structures. Working over a ring, one would have to take a bit more care in making sure T and the a_i were invertible, but this would not be hard.

The feature that mandates a little extra care is the same one that presents an interesting possibility here — rings can have nontrivial ideals. Consider the case $R = \mathbb{Z}/n\mathbb{Z}$, $A \in GL_m(R)$, $x, y \in R^m$, and $Ax = y$. Here, though A is invertible, many of its entries may not be, and multiplication by one of the non-invertible (in this case, we might equivalently say 'even') elements of R necessarily results in a loss of information. Many such multiplications happen in $Ax = y$, yet all of the information that is 'lost' in a local sense is preserved elsewhere in the system, because the operation is invertible.

This property could have interesting implications, perhaps forcing an attacker to guess more simultaneous plaintexts than are already necessary, or at least accelerating the branching in the cycle game. Using direct products of multiple rings could have structural and computational advantages, but only if an operation were used in encryption that caused the individual coordinates of their elements to interact, so that the system could not be pulled apart into several smaller ones each acting on one coordinate.

6.5 Beyond Affine

Here the public key differs from the private key only by the affine functions 'interjected' between the f_i and the g_i. Could a broader and more complex family of functions be used for added security (and a larger key space)?

In other words, given

$$e_i(x, y, z) = g_i\left(t_{s_{i,1},i} f_{s_{i,1}}(x) + t_{s_{i,2},i} f_{s_{i,2}}(y) + t_{s_{i,3},i} f_{s_{i,3}}(z)\right)$$

as before, what functions $h_i, h_{i,1}, h_{i,2}, h_{i,3}$ could be used such that

$$g'_i(x) = g\left(h_i(x)\right) \text{ and } f'_{i,j}(x) = h_{i,j}\left(f_{s_{i,j}}(x)\right)$$

for $1 \leq j \leq 3$ while preserving

$$e_i(x, y, z) = g'_i\left(f'_{i,1}(x) + f'_{i,2}(x) + f'_{i,2}(x)\right)$$

as before? It seems likely that going beyond the affine case would require that the the f_i and g_i be used in defining the h functions, as to work in general h_i needs to 'split' as only affine functions can. A good method for taking the f_i and g_i and generating non-affine h functions would both add difficulty to future cryptanalysis and be mathematically interesting in its own right.

6.6 Layered Encryption

After a round of encryption, each ciphertext word depends on exactly three plaintext words. If one were to then encrypt the ciphertext again - either with the same public key or with another - the resulting second ciphertext would have words which depended on up to nine of the original plaintext words. Some might repeat, making the 'up to' necessary, but by combining complementary matrix shapes this effect could be minimized or, for sufficiently large m, avoided entirely.

After a few rounds of encryption, one could achieve full dependency in the sense that every word of ciphertext depended on every word of plaintext. This certainly sounds like a desriable quality and it would make the attacks in chapter three no better than brute force.

However, this could not be accomplished naively by simply sending out several public keys and instructing the recipient to use them in turn. In that case, an attacker would just attack them one at a time, in reverse order.

To gain significant security from layered encryptions, one would have to combine these several public keys into one key which gave the recipient the ability to compute the final ciphertext without giving the ability to compute the intermediate ones. Using methods analogous to those in the first chapter would give an unusably large key. If there is a method that keeps public key size reazonable without giving away private key information, it could result in an even more secure system and reduce or even remove the graph-theoretic considerations.

6.7 The Quantum Question

Several quantum algorithms are known which are significantly better than their classical counterparts [11], and none are more famous than those that compromise today's most important public-key cryptosystems. These algorithms tend to gain their advantage from a few operations which quantum computers perform exceptionally well, such as the discrete Fourier transform. So far, it does not appear that quantum computers are better at solving the problems relevant to breaking this system than are classical computers.

As research accelerates, it is possible that this could change. It would not take a tremendous number of qubits to hold a superposition of all the possible a_i and $b_{i,j}$. Whether an algorithm exists which could operate on this qubit string with a high probability of returning the correct a_i and $b_{i,j}$, is beyond the scope of this paper At best, we can use heuristics to deem it unlikely due to the lack of exploitable structure. Then again, someone could conceivably design a classical algorithm to solve the same problems while cleverly avoiding the pitfalls outlined in chapters two and three. Really, it's impossible to say.

Bibliography

[1] Berma, G. P., Doolen, G. D., Mainieri, R., and Tsifrinovich, V. I. *Introduction to Quantum Computers*. World Scientific, 1998.

[2] Biggs, N. *Constructions for cubic graphs with large girth*. PhD thesis, London School of Economics, 1997.

[3] Brown, E. The many names of (7 3 1). *Mathematics Magazine* (2002).

[4] Cohen, H. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1993.

[5] Dummit, D. S., and Foote, R. M. *Abstract Algebra*. John Wiley and Sons, 2004.

[6] Elgamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE TRANSACTIONS ON INFORMATION THEORY* (1985).

[7] Hankerson, D., Menezes, A., and Vanstone, S. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[8] Hungerford, T. *Algebra*. Springer-Verlag, 1974.

[9] Leon, S. J. *Linear Algebra with Applications*. Pearson Prentice Hall, 2006.

[10] Marcus, D. A. *Graph Theory: A Problem Oriented Approach*. Mathematical Association of America, 2008.

[11] Mermin, N. D. *Quantum Computer Science*. Cambridge University Press, 2007.

[12] Piper, F., and Murphy, S. *Cryptography: A Very Short Introduction*. Oxford University Press, 2002.

[13] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., and Ferguson, N. Twofish: A 128-bit block cipher. In *in First Advanced Encryption Standard (AES) Conference* (1998).

[14] Shor, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.* (1997).

[15] Smart, N. *Cryptography: an Introduction*. McGraw-Hill, 2003.

[16] Stinson, D. R. *Cryptography: Theory and Practice*. CRC Press, 2002.

[17] Trappe, W., and Washington, L. C. *Introduction to Cryptography with Coding Theory*. Pearson Prentice Hall, 2006.

[18] West, D. B. *Introduction to Graph Theory*. Prentice Hall, 1996.

[19] Yaschenko, V. V. *Cryptography: An Introduction*. American Mathematical Society, 2002.