Toy Public Key Cryptography

Jiri Lebl

May 12, 2003

1 Introduction

In this paper I will discuss two toy public key cryptosystems, analyze them, and compare them to some real cryptosystems. I will also discuss certain aspects of digital signature schemes with respect to these cryptosystems. The first toy cryptosystem is the Larry-Curly-Moe cryptosystem which is an attempt at a different use of the knapsack problem. The second is the Easy-Discrete-Log cryptosystem which makes use of an easy to solve discrete log problem. Information about real systems is taken from Handbook of Applied Cryptography [6], Koblitz's A Course in Number Theory and Cryptography [5] and Coding Theory and Cryptography [4].

2 Knapsack Based Cryptosystems and the Larry-Curly-Moe System

The knapsack problem is an NP complete problem which tries to solve the subset sum problem. Given a set of integers $S = \{x_1, x_2, \cdots, x_n\}$ find a subset s which sums to a certain integer. This problem seems appealing to cryptography since it deals with integers, meaning it is easy to embed messages, and since it is known to be very hard to solve in the general case. Since it is an NP complete problem, it could very well be polynomial time solvable, but more likely it will not have a polynomial time solution. Even if a polynomial time solution would exist, this is saying something about the asymptotic behavior of the problem, and may in fact not have any relevance to how hard it is to solve for useful sizes of n.

Just the fact that a problem is NP complete doesn't give any indication of it's usefulness however since again this is the asymptotic behavior, and it may very well be that for practical n this problem is easily solvable, even if the asymptotic behavior is not polynomial. The knapsack problem is however hard to solve in the genearl case for fairly small n.

Let's now look a the *density* of the knapsack. HAC [6] defines the density as

$$d = \frac{n}{\log_2 \max\{x_i\}}.$$

The density of the knapsack must be less then 1 for there to be a unique solution to the problem. In general a fairly brute force approach to solving this system is by the Meet-in-the-middle [6] algorithm and this has complexity of about $O(n2^{n/2})$ which is pretty inefficient. However if the density is low then an algorithm of complexity about $O(Cn^4)$ (where C depends on the size of the integers) can be used, which is a problem of finding a short vector in a lattice. This attack is not always successful, but according to [6] this is typically successful if the density of the knapsack is less then 0.9408.

A super-increasing knapsack is a special case in which every integer in the knapsack is at least twice the previous integer. This is easily solved by a greedy algorithm and is of complexity $O(n)$. One idea of using knapsacks is to somehow disguise such a knapsack so that it is no longer super-increasing. A cryptosystem which uses this idea is the Merkle-Hellman cryptosystem which takes a super-increasing knapsack and disguises it by multiplying by an invertible number modulo some large modulus. The resulting knapsack will still have lower then 1 density and thus be uniquely solvable, but it will no longer be super-increasing. A sender will just translate his message to binary, match up each binary digit with a number in the knapsack, sum numbers corresponding to 1's and send this to the receiver, who knows the modulus and the disguising element and can then convert this number to something that can be solved by the original super-increasing knapsack.

Unfortunately, one does not need the original modulus and the disguising element to get a super-increasing set that also works. There in fact exists [6] a polynomial time algorithm for doing this. Even if this was not the case, the problem is that if the density must be less then 1, then there exists the low density attack. So even if we were to improve the disguising of the knapsack, or in some other way had a low density knapsack which the receiver can easily solve, then the attacker still has an attack of complexity $O(n^4)$ at his disposal.

So in order to have a secure scheme using the knapsack we somehow have to utilize knapsack of density higher then 1. The only such useful cryptosystem is the Chor-Rivest cryptosystem [6]. I will only give a short overview of the idea behind this cryptosystem rather then the details. In this cryptosystem, you generate a knapsack of length p (where p is a prime, or it could in fact be a prime power), such that if you sum any h (where $h < p$) numbers the receiver has an easy method to figure out which numbers were summed. Now notice that this knapsack no longer needs to be uniquely solvable in general, but only in case where exactly h numbers are summed together. For suggested parameters $p = 197$ and $h = 24$ we get that the knapsack density is about 1.077 [6], thus thwarting the low density attack. This system has an advantage of very fast encryption, but has the drawback of very large public key which for the suggested parameters is about 36000 bits.

Now finally I present the Larry-Curly-Moe encryption, which can utilize a high density knapsack, but has the disadvantage of being incredibly inefficient and in fact being reducible to the RSA problem, thus giving no advantage over RSA. So in short I present the RSA problem: Suppose that there is some $n = pq$ where p and q are prime and we know $m^e \pmod{n}$ for some e invertible mod

$\phi(n)$, can we find m given that we don't know p and q.

The idea behind Larry-Curly-Moe is that we can randomly construct a knapsack for which we have one particular solution. So first pick $n = pq$ where p and q are primes and n will be public while p and q are private. Next construct a random knapsack $S = \{x_1, x_2, \cdots, x_n\}$ such that a certain subset sums to $lcm(p-1, q-1)+1$, say $\sum_{i=1}^{r} x_{k_i} = lcm(p-1, q-1)+1$. This knapsack will then be our public key together with n. Now the sender will take the message m that he wishes to send and compute $c_i = m^{x_i} \pmod{n}$ for every x_i in the knapsack and send those numbers to the receiver. The receiver computes

$$\prod_{i=1}^{r} c_{k_i} \equiv \prod_{i=1}^{r} m^{x_{k_i}} \equiv m^{lcm(p-1,q-1)+1} \equiv m \pmod{n}$$

The first attack is to factorize n in which case we could just reduce mod p or q any one of the c_i and then compute the appropriate root in $\mathcal{F}_p$ or $\mathcal{F}_q$.

The next attack on this system is to find the gcd of all the elements of S and thus get a linear combination of the elements of S (or some subset). If this gcd is 1, then we can just take the appropriate powers of the encrypted numbers and multiply them together to get m back. That is suppose $1 = \sum a_i x_i$, then we can just compute $\prod c_i^{a_i} = m^{\sum a_i x_i} = m$ (all mod n). One way to avoid this attack is would be to make sure that the gcd is greater then 1. We can do this by adding $lcm(p-1, q-1)$ to any of the x_i any number of times until we get a common factor. This way the subset will sum to $s \times lcm(p-1, q-1)+1$ for some integer s, which would still give m back after decryption. For example we could take the lowest prime u, not dividing $lcm(p-1, q-1)$ and then we can make a new knapsack by adding at most $u-1$ times $lcm(p-1, q-1)$ to each element in the knapsack until this element is divisible by u.

An attack on any particular c_i can be made not equivalent to the RSA problem. It is a problem of finding dth roots in $\mathbb{Z}_n$, where d is not necessarily invertible mod $\phi(n)$. The problem of finding the square root is for example equivalent to factoring n [6], while RSA is not proven to necessarily be as hard. However if we consider that we have all the c_i for a particular m available then we can again get the gcd of all the x_i and again compute the linear combination of x_i that gives the gcd and then we will get $m^u \pmod{n}$, since above we made sure that the knapsack has gcd of u. But u is then invertible mod $\phi(n)$ since it does not divide it (it doesn't divide $lcm(p-1, q-1)$). And so we have reduced this to an RSA problem. We note that since the knapsack must contain a combination that gives $s \times lcm(p-1, q-1)+1$ for some integer s, and so the gcd must not divide $lcm(p-1, q-1)$ and thus must always be invertible mod $\phi(n)$. This means that this system is no more secure then RSA and is likely less secure given that we provide quite a bit of information to the attacker, as we provide the message raised to many different exponents.

This last attack can be generalizable to an attack on RSA if the same modulus were to be used by several people. If both Alice and Bob are using modulus n, but different exponents e_a and e_b and by chance $gcd(e_a, e_b) = 1$, and I send them both the same message m, then an attacker can recover the message in the same way as above. This means that for security reasons, each user should pick a different modulus.

3 Provably Secure Systems, Rabin System and the Easy-Discrete-Log System

Each public key cryptosystem has an associated hard problem. It would be nice to have a proof of how hard this problem really is. Take for example RSA. If we can factor the modulus n, then we can solve the RSA problem, however the converse is not proved. There is a possibility that a systematic way to recover messages exists without being able to factor n [4].

There however exist cryptosystems which are provably as hard as factoring n. One such cryptosystem is the Rabin scheme which uses the square root modulo $n = pq$ [4]. The Rabin hard problem is this: suppose you are given the numbers c and n where $c \equiv m^2 \pmod{n}$ and $n = pq$, find m. Now suppose that you had a polynomial time algorithm to solve this. Then it can be shown that you can factor n as follows. Take any x and compute $c \equiv x^2 \pmod{n}$, then run the algorithm to get a square root of c mod n and call this y. Since there are 4 different square roots, then with probability $\frac{1}{2}$, $y \not\equiv \pm x \pmod{n}$, in this case we have that $x^2 \equiv y^2 \pmod{n}$ or $x^2 - y^2$ is a non-zero multiple of n. In this case we have $x^2 - y^2 = (x+y)(x-y)$. Next we just take the $gcd(x-y, n)$ to recover a non-trivial factor.

However if we know the factorization of $n = pq$, then we can solve the square root problem given that we can solve the square root problem modulo p and modulo q. There exists a randomized polynomial time algorithm for doing this for all primes p, and there exist deterministic algorithms for special primes p such as $p \equiv 3 \pmod{4}$ [4] [6].

So the Rabin scheme is to send $c = m^2 \pmod{n}$ and the receiver, knowing the factorization of n can then recover the four possible square roots. The one obvious drawback to this scheme is that the receiver has to decide which root is the correct one, which has to be decided depending on the context. Because of this drawback, this scheme is susceptible to the chosen ciphertext attack [6]. Suppose an attacker has the ability to at some point decipher several messages, perhaps by temporarily gaining access to the deciphering equipment. In this case the attacker could just choose an x, square it and have it decrypted. The naive decryption mechanism returns the 4 square roots as it expects the user to pick the right one. The attacker can then factorize n in the same way that was given above for the proof that the Rabin problem is as hard as factoring. One way suggested [6] to remove this problem is to add some redundancy to the message, such that with high probability, the decryption mechanism will recognize the correct root out of the four, or fail in case no root has this redundancy. This way the attacker no longer gets any useful information. However in this case the proof of equivalence to factoring no longer holds since we are no longer dealing with arbitrary square roots [4].

One modification to the Rabin scheme which is still provably as secure as factoring $n = pq$ and does not have the chosen ciphertext problem is a modification proposed by Williams [4]. The only restriction here is that $p \equiv q \equiv 3 \pmod 4$. The ciphertext will contain two more bits of information which will allow the receiver to identify the proper root, and the public key will contain one more integer. The scheme works as follows. After selecting $n = pq$ we select s such that $\left(\frac{s}{n}\right) = -1$, then (n, s) is the public key. We also compute $d = \frac{(p-1)(q-1)/4+1}{2}$ and that is private. Let m be the message and assume that $gcd(m, n) = 1$, then if $\left(\frac{m}{n}\right) = -1$ then let $b_1 = 1$ and $m_0 = sm \pmod n$, if $\left(\frac{m}{n}\right) = 1$ then let $b_1 = 0$ and $m_0 = m$. Now let $b_2 = m_0 \pmod n$ and $c = m_0^2 \pmod n$. The sender then sends (c, b_1, b_2). Now it can be shown that if $\left(\frac{m_0}{n}\right) = 1$ then $m_0 \equiv \pm c^d \pmod n$, which means the receiver can recover $\pm m_0 \pmod n$ and we can distinguish between the sign with b_2 and thus recover the m_0. Then we can recover m by $m \equiv s^{-b_1} m_0 \pmod n$.

It is interesting that the Rabin scheme is not as widely used as the RSA scheme, even though it appears to be an easier to work with scheme, even including the modification by Williams. Furthermore the efficiency of encryption is better then RSA since encryption only requires one modular squaring. For the modification by Williams, we must calculate $\left(\frac{m}{n}\right)$, so this is no longer as efficient. Still it is interesting that this scheme is not used in practice as often, especially given that there exists a proof about its hardness.

We can now turn to our toy example of a scheme that is provably at least as secure as factoring, this is the Easy-Discrete-Log scheme. While this technique will use the discrete logarithm over finite fields, this in fact is not the hard problem that we will be looking at. So suppose that you have $n = pq$, and further suppose that there exists an easy solution of $\log_g c \pmod p$, where g is a generator of $\mathbb{Z}_p^*$. Also let $M < p - 1$ be some number small enough such that you cannot discover p, but still close to about $\sqrt{n}$ for efficiency. Now let (n, g) be the public key and M be a standard number for all n of certain range. A sender can then send messages m where $0 \leq m \leq M$ by computing $c \equiv g^m \pmod n$ and then sending c. The receiver then reduces c mod p and then we know that $c \equiv g^m \pmod p$, since $m < p - 1$ this is still the smallest such m. We can then recover m by computing $m = \log_g c \pmod p$. This system is provably as secure as factoring n. This is because if we can solve the problem of finding logs in $\mathbb{Z}_n$ for n composite we can factor n [6].

The only problem is how to make the discrete log problem easy. My first idea was to choose p such that $p - 1$ is a B-smooth number, for some reasonably small B, then the Silver-Hellman-Pohlig algorithm could be used to relatively quickly recover the message. However then the Pollard $p - 1$ factoring method could be used [6]. The computational complexity of the Silver-Hellman-Pohlig algorithm for p is $O(\sum e_i(\log_2(p-1) + \sqrt{p_i}))$ where p_i are the prime factors of $p - 1$ and e_i are the corresponding exponents. Given that the exponents are relatively small compared to the primes then this is really somewhere on the order of $C\sqrt{B}$ where C would be some relatively small integer. On the other hand the Pollard $p - 1$ algorithm has computational complexity of $O(B\frac{\ln n}{\ln B})$.

This means that we must choose B pretty high in order to foil the Pollard $p-1$ algorithm. Asymptotically the Silver-Hellman-Pohlig algorithm is a lot faster then the factoring using the Pollard $p-1$ algorithm. But the advantage here is not that great. Still factoring n is provably the only passive attack on this system.

So scraping the idea of carefully picking p, let's look at index calculus style algorithms for computing logarithms. The advantage of these algorithms is that once the precalculation is done, then given we have chosen our factor base properly then the calculation of many logarithms for that particular base in that particular field is very quick. So we could compute our factor base once and keep it around as part of our private key. Now according to [6], the best index calculus based methods are about the same complexity as the best factoring methods. But remember that we are working with only half the bits then the attacker, since we are looking at p and the attacker is looking at n. With these algorithms we can get running time of $O(\exp((c+o(1))(\log p)^{1/3}(\log\log p)^{2/3}))$ for c around 1.923. Same running time is for the best factoring methods if we replace p with n. It should be feasible [6] to use these methods for primes up to about 332 bits and this was said in 1990. We should be able to do better nowadays, though this requirement probably makes this cryptosystem impractical. However even if this calculation should take a long time, this is only for the private key which is only generated once.

In summary, this cryptosystem has fairly small public key which is about the size of n (the generator can be very small), and the rate is about $\frac{1}{2}$. Furthermore encryption is reasonably fast, and decryption is reasonably fast given we have a good factor base. The disadvantages of the system is the hardness of making up the private key, which is the precomputation step of the index calculus algorithm, and the size of the private key since the precomputed data for the factorbase is likely to be very large. Another disadvantage is that we are more limited in how large the public key is. With this system having n be about 2000 bits is unfeasible since then p would have about 1000 bits and that would make discrete logs too hard to compute. Meanwhile with the RSA or Rabin scheme we can increase the public key much further without making the public key generation all that much harder.

Unfortunately this system is also susceptible to a chosen-ciphertext attack. Suppose we gain access to the decryption function, and start feeding it $g^{kM} \pmod{n}$, for $k = 1, 2, 3, \cdots$. Either the decryption function will refuse to return anything since the message is not a legal message (it is greater then M) or it will keep returning kM until at some point it will "roll over" to some smaller value. From this we could then compute $p-1$, since the returned exponents will be all modulo $p-1$. We could protect against this attack in two ways. First we could choose M substantially smaller then p, but this would in turn lower the rate substantially as well. Another way to protect against this attack is to also have a factorbase precomputed for q and calculate not only the discrete log mod p but also log q (supposing that g is also a generator of $\mathbb{Z}_q$). Then when the exponents "roll over" the answer we get mod p will not be equal to the answer we get mod q and the decryption function can in that case fail and provide no output. In this case it should also provide no output when the decrypted message is anything greater then M of course. However since RSA is also susceptible to an adaptive chosen ciphertext attack [6] and it is still being used in practice, then I suppose this is not as an important issue.

If we can also solve the discrete log problem over $\mathbb{Z}_q$ then we can with little effort solve an arbitrary discrete log over $\mathbb{Z}_n$, and thus have messages in the range of 1 to $\phi(n)$ and a rate close to 1, but then we'd again be susceptible to the chosen ciphertext attack supposing that M is "close enough" to $\phi(n)$.

As a further generalization, this attack may be generalized to polynomials and thus use the discrete log problem in $\mathbb{F}_q$ for q a prime power. For example let $n(x) = p(x)q(x)$ where p and q are irreducible polynomials over the base field. Then the message m can be sent again just as previously by $c(x) = g(x)^m \pmod{n(x)}$ where $g(x)$ is a generator for $\mathbb{F}/p(x)$. Then the receiver again reduces mod $p(x)$ and computes the discrete log. This generalization may not however be as good as there are fairly good polynomial factorization algorithms, for example if the base field is $\mathbb{Z}_p$ then there exists an $O(n^2)$ algorithm [6]. Again this would require more research into discrete log problems over different finite fields, factoring polynomials over different fields.

4 Comparison of Parameters for Real World Application

In this section I will assume the position of a would be user of a public key encryption scheme with a proposed usage on the internet for key exchange. I will assume that I have a symmetric key system that uses a fairly short key, let's say about 256 bits, and I want to run a short session over IP [1] network and let's further assume that I wish to use TCP [2] to run the session.

So let's first look at rate. An IP packet header is 160 bits long and a TCP packet header is 192 bits long. Let's further assume that we can send up to 1kb (kilo byte, so 8192 bits) of data for each packet without fragmenting. And let's ignore the overhead of establishing a connection. Let's call A the sender and B the receiver in terms of the public key. Each packet will have an overhead of $160 + 192 = 352$ bits. Let's assume that B somehow knows that A wants the public key and will just send it when the session starts. To send the public key B will have to send 352 bits per packet (overhead) and can send up to 1kb (about 8000 bits) of data per packet if we limit ourselves this way. Then A will send the encrypted session key (for the symmetric key system) to B. This will be 256 bits of data but it depends on the cryptosystem rate to see how much we can send. So in one packet we can send about 8000 bits which corresponds to about $\frac{1}{32}$ rate. So let's compare some of the cryptosystems discussed with respect to this. For systems which depend on factoring n let's use n of size 1024 bits. For all of RSA, Rabin, Rabin-Williams, and Easy-Discrete-Log we can send 256 bits in one go. In fact it does not matter that RSA has rate 1 and Easy-Discrete-Log has rate 1/2. For all we care the rate could be 1/4 for this rate of security. For Larry-Curly-Moe we will assume that the size of the knapsack is 100 and so we will send $100 \times 1024 = 102400$ bits of data for which we will need 13 packets. For the Easy-Discrete-Log system and the Rabin-Williams scheme we will assume we can fit the second number of the public key into 8 bits. For Chor-Rivest we will use the suggested parameters discussed above and thus have a public key of 36 thousand bits, which means we will use 5 packets. The Chor-Rivest cryptosystem for $p = 197$ and $h = 24$ will have rate of about $\log_2 \binom{197}{24} / \log_2 197^{24} \approx 0.555$. That is for each 101 bits of data we send 183 bits of ciphertext. So this means that to send 256 bits we will need to send $183 \times 3 = 549$ bits. We summarize our result in Table 1. The first two columns, the $A \to B$, basically represent the "data" part, that is sending the 256 bit session key, and the $B \to A$ communication is sending the public key. It should be noted that the public key can be cached and then we should only look at the first two columns. In this case the Chor-Rivest is actually the most efficient since we will only send 901 bits of network data (plus any padding bits of course, since we can't really send an odd number of bits). This is kind of interesting since we can see that as long as the rate is not too horrible, then for these real world parameters, the rate is irrelevant. What made the Chor-Rivest scheme best is that the data can be sent in smaller chunks. We are wasting 768 bits of data with RSA, but we can't really make this any smaller since that would reduce security. So data rate of a cryptosystem should be viewed in terms of actual usage. For sending random data it may be relevant, but if this is only used for key exchange, then the data rate of the system will likely be irrelevant.

Scheme	$A \to B$ (packets)	$A \to B$ (bits)	$B \to A$ (packets)	$B \to A$ (bits)	Total Packets	Total Bits
Chor-Rivest	1	901	5	37760	6	38661
RSA	1	1376	1	1376	2	2752
Rabin	1	1376	1	1376	2	2752
Easy-Discrete-Log	1	1376	1	1384	2	2760
Rabin-Williams	1	1378	1	1384	2	2762
Larry-Curly-Moe	13	108000	13	106976	26	214976

Table 1: Comparison of Rate

So perhaps the data rate would not be the deciding characteristic, though it may play a role (it would certainly eliminate the Larry-Curly-Moe system). The public key size seems to be a much larger problem unless we assume caching is used a lot, which we cannot assume for random connections, but only for connections between two hosts that talk to each other a lot. Supposing random connections, I now have to choose from RSA, Rabin, Rabin-Williams or Easy-Discrete-Log, as they are all about equal in how much data I send over the network for my application. So I would now look at the performance. For generating the private key, RSA, Rabin and Rabin-Williams are about the same since this basically reduces to finding 2 random large primes of the right size and nice properties. For the Easy-Discrete-Log system this requires precalculation of the factorbase and this may or may not be prohibiting. The performance that we really wish to look at is the encryption and decryption speed. The more relevant of these is likely the encryption speed as this will be done by client computers rather then larger servers if we assume this will be used for commerce. For RSA it is common [6] to use encryption exponent 3 or $2^{16} + 1 = 65537$ to make encryption fast. Still this will be at least one modular squaring and one modular multiplication. The Rabin scheme on the other hand is one modular squaring. Rabin-Williams and Easy-Discrete-Log systems will do a fairly random modular exponentiation and thus will be more intensive computationally.

Another factor here is security. Both Easy-Discrete-Log and Rabin-Williams are provably as secure as factoring n. If we assume that Rabin uses redundancy in the Rabin scheme then we no longer can prove this level of security, but intuitively it seems like it should be just as secure. RSA is not provably secure but has undergone a lot of analysis and so it appears to be as secure as factoring n.

Based on these parameters and the above analysis, I would probably choose the Rabin scheme (using redundancy). While not provably secure, it seems very unlikely that the redundancy (depending how it is done) will make it somehow easier to solve the Rabin hard problem. Furthermore it is easier to encrypt using this scheme then it is with the RSA scheme, and decryption is just about as expensive as RSA, depending on the primes that we choose. Of course I have only compared a few systems, mostly with the goal of comparing these to Larry-Curly-Moe and the Easy-Discrete-Log system. Judging from the previous discussion, the Easy-Discrete-Log system actually fares fairly well against real systems aside from the very hard private key generation.

5 Digital Signatures with RSA and Easy-Discrete-Log

First I will discuss the RSA scheme for digital signatures. It is possible to use many public key cryptosystems in a similar way for digital signature. The idea is that to sign a message, the signer (who has the private key) first decrypts the message, and then anyone who has the public key can check that this decryption is correct by encryption to obtain the message. Of course it is a little bit more complicated. There are two types of signature schemes [6], the first are signature schemes with appendix and the second are signature schemes with message recovery. Signature schemes with appendix are schemes where you cannot recover a message from the signature. That is, with these schemes you can only check that some message corresponds to some key and signer. For schemes with message recovery, you get both the message and the fact that it comes from the signer.

So first let's explain how schemes with message recovery work, and let's use RSA as an example. So we have $n = pq$, and so (n, e) is our public key and

(p, q, d) is our private key (where $(m^d)^e \equiv m \pmod{n}$). We have a message space $\mathcal{M}$ and a signing space $\mathcal{M}_\mathcal{S}$, we also have a $1-1$ map R from $\mathcal{M}$ to $\mathcal{M}_\mathcal{S}$ which is called the redundancy function. This function is publicly known and so can be considered part of our public key. The image of R is called the $\mathcal{M}_R$. Here $\mathcal{M}$ and in turn $\mathcal{M}_R$ is much smaller then $\mathcal{M}_\mathcal{S}$, so we will be able to send much smaller messages then with RSA encryption. So we take our message $m \in \mathcal{M}$ and encode it with R to some element in $\mathcal{M}_R$ which is in turn a subset of $\mathcal{M}_\mathcal{S}$. Now for RSA, the $\mathcal{M}_\mathcal{S}$ is basically all the integers modulo $n = pq$. We then use RSA to "decrypt" using d. A person wishing to recover the message and verify our signature just "encrypts" using e to get an element of $\mathcal{M}_\mathcal{S}$. If this element is not in the image of R, then this message is bogus and should be rejected as a forgery. The inverse of R can then be used to recover the message. Note that this means that the image of R should be a lot smaller then the whole $\mathcal{M}_\mathcal{S}$, so that a forgery can be detected. So in more mathematical terms the signer has a message m and does $\tilde{m} = R(m)$, then the signer computes $s = \tilde{m}^d \pmod{n}$, and sends the signature s. The receiver then computes $\tilde{m} = s^e \pmod{n}$, verifies that $\tilde{m}$ is in $\mathcal{M}_R$ and if so computes $m = R^{-1}(\tilde{m})$ to recover the message.

Of course this whole thing relies on the fact that we trust that the private key that signed the message m actually belongs to the user that we believe sent the message. This means that there is still the problem of verifying that the public key for this signature is really the one of the the sender and not some active adversary in the middle. In practice what is done is that key-signing sessions are held (something fairly common at computer conferences) where people sign each other's public keys with their own if they can verify that they've got the right person. Then when you obtain a public key, you can also check that it has been signed by many other people, some of whom you may trust. Usually this information is kept on public key servers such that it is easy to recover for anyone. Also if a private key is compromised, the original holder can invalidate the key. An explanation of how this works with PGP software can be found in [3].

Similar attacks exist on the signature scheme of RSA as there exist on the encryption scheme [6] using RSA since the same concept is at work really. The issues surrounding signature schemes are a little different however. An adversary may obtain a valid signature for some message only, perhaps he can't choose the message arbitrarily, but perhaps just from some set or previously sent messages. For example an adversary may resend a message later that we sent earlier and the receiver would still think it came from us. Some of these types of attacks can be prevented by protocol. There are also mathematical issues at play. The choice of the function R above is for example very important. One problem of the RSA scheme is its multiplicative property [6]. Suppose that $s_1 \equiv m_1^d \pmod{n}$ and $s_2 \equiv m_2^d \pmod{n}$, then we have $s_1 s_2 \equiv (m_1 m_2)^d \pmod{n}$. So if it were the case that $R(a)R(b) = R(ab)$ for many (essentially all) a, b in the message space, then we could really create many new signed messages out of a few captured signed messages.

If we were to convert the Larry-Curly-Moe into a signature scheme, it would be essentially the same as the RSA scheme, so let's look at the Easy-Discrete-Log scheme. It is not as simple as with RSA, since the encryption and decryption here are really not simple commutative operations. We will design a signature scheme with appendix. This means that the receiver cannot recover the message from the signature, and must have gotten the message in some other way. The receiver can just check that the message actually originated from us. Suppose that we have a private key for a modified Easy-Discrete-Log system, that is, we have an integer $n = pq$ and we can solve discrete logs to base g in both $\mathbb{Z}_p$ and $\mathbb{Z}_q$ quickly, which in turn will enable us to solve discrete logs to base g in $\mathbb{Z}_n$. Next suppose that we have a hash function h which takes the message and returns a number between 1 and n, and this hash function is public. If we wish to send a signed message, we first take our message M and apply the hash function to it to get $m = h(M)$. Then we calculate $s = \log_g m \pmod{n}$ by first calculating $s_p = \log_g m \pmod{p}$ and then $s_q = \log_g m \pmod{q}$ and then calculating s from this. It is very unlikely that m is a multiple of p or q so we will ignore this error case (the h function could be modified to prevent this anyway). Now the receiver takes the message M and obtains $m = h(M)$. Then the receiver proceeds to calculate $g^s \pmod{n}$. If $m \equiv g^s \pmod{n}$ then the signature checks out, otherwise it is a fraud.

Basically we are demonstrating to the receiver that we can take discrete log in $\mathbb{Z}_n^*$ which is provably at least as hard as having factored n [6]. This means that the only person who could have done this is someone who actually owns the private key. Of course our hash function should again satisfy some constraints. For one it should not be multiplicative since the Easy-Discrete-Log scheme is multiplicative in the sense that $m_1 m_2 \equiv g^{s_1} g^{s_2} \equiv g^{s_1+s_2} \pmod{n}$. An adversary could take two signatures, add them together, multiply the messages and get a new message that seems signed.

6 Conclusion

I have looked at two toy examples of public key schemes and compared them with some similar existing schemes. Even very different schemes face many similar issues in practice. I have also presented some evidence that the second scheme, the Easy-Discrete-Log, while impractical in the step of key generation can in fact compete in parameters with real world systems. It would merit more research into discrete logs over finite fields. The problem of performing the precalculation step may not be so hard since we are really free to pick g, p and the factorbase for the index calculus algorithm all at the same time, while in the general problem we are already presented with a g and a p. While I don't see how we may take advantage of this fact, it may be possible. So while what I have presented deals with a general discrete log solution, it may very well be that we do not need to solve the general problem at all.

References

[1] RFC 791, Internet Protocol DARPA Internet Program Protocol Specification. http://www.freesoft.org/CIE/RFC/791/index.htm, 1981.

[2] RFC 793, Transmission Control Protocol DARPA Internet Program Protocol Specification. http://www.freesoft.org/CIE/RFC/793/index.htm, 1981.

[3] Patrick Feisthammel. Explanation of the web of trust of PGP. http://www.rubin.ch/pgp/weboftrust.en.html, 2002.

[4] D. R. Hankerson, D. G. Hoffman, D. A. Leonard, C. C. Lindner, K. T. Phelps, C. A. Rodger, and J. R. Wall. *Coding Theory and Cryptography.* Marcel Dekker, New York, New York, 2000.

[5] Neal Koblitz. *A Course in Number Theory and Cryptography.* Springer-Verlag, New York, New York, 1994.

[6] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, Florida, 1997.