Encryption, Decryption, and Cracking Techniques

MAT258: Spring 2015
Dr. Brigitta Vermesi

Authors
Chase Hutchens
Scott Smith
Joshua Shlemmer

One of the first things that we wanted to study was the process of decomposing particular integer numbers, prime or otherwise into two prime factors that compose the particular number we are attempting to prime factorize. Some of the process that is composed within this method revolves around conjecture instead of absolute proof.

When beginning our search for valid primes that may make up the number, the first thing that is done is to obtain a minimal prime that when multiplied with itself will produce a result slightly larger than the number that we are prime factorizing. This is obtained by : $p' = \lceil \phi \sqrt{F} \rceil$ where we used the value $\phi = \frac{1+\sqrt{5}}{2}$ and $F = ToFactor$. The reason for choosing to scale by the golden ratio is resulted from conjecture and the ideal that the golden ratio and prime numbers go hand in hand. Also, when we were searching for one of our particular prime factorization values, a number close to two was needed to be able to obtain the missing prime that was needed for utilization of determining if the number could be properly factored. Continuing on, we repeat this previous process for *p'* to obtain : $p'' = \lceil \phi \sqrt{p'} \rceil$. We iterate through checking if the condition : $p' \, mod \, p'' \neq 0$ holds true, in which if it does hold true we decrement $p'' = p'' - 1$, the iteration is terminated either if the condition is false or if $p'' = 1$. If $p'' = 1$ holds true, we know we have found a potential prime factor. After the iteration if we have determined *p'* to be a valid prime, we store it as a possible prime that can be used in our prime factorization. The main discrepancy with this algorithm is that the prime factors will be contained within the range : $2 \leq p' \leq \lceil \phi \sqrt{F} \rceil$.

After we have determined all possible primes that may make up our number we are prime factorizing we must iterate through multiplying each prime with another checking to see if two prime factors will produce our original number : F . Which if we have n total primes that we have found, we would need to do a maximum of : $n!$ combinations before finding our correct factorization. In hindsight, this number could be greatly reduced since we are only looking for valid prime factors that result when multiplied together within: $10^{\log \lfloor F \rfloor} \leq F \leq 10^{\log \lfloor F \rfloor + 1}$.

Our next choice was to utilize prime numbers in conjunction with the RSA cryptosystem algorithm. (http://en.wikipedia.org/wiki/RSA_%28cryptosystem%29) The first thing that needed to be done was to convert our textual data into two digit number representations. Beginning at the value 'A', it is converted to the number : 01 to 'Z' which is equivalent to : 26, followed by other symbols and numbers up to the '~' which is valued at : 58. Making sure each unencrypted data portion has two digits per piece is the most important part for our algorithm.

While implementing the RSA algorithm the first obstacle encountered was how to deal with the large numbers after the data to be encrypted is converted to it's equivalent number representation. This is where we then decided to convert those pieces into individual blocks, padded at *n* values of 2 digits each, such as the values : |401 | 2001 | corresponds to the text : “Data”. This particular example consists of padding at 2 digits of 2, which is our default, the values are read from right to left. However, as the padding size increases more discrepancies appear, producing invalid decryption results which is associated with the number becoming too large and re-wrapping.

(http://en.wikipedia.org/wiki/Modular_exponentiation) Due to the RSA algorithm needing to calculate values in the form: $c \equiv b^e (mod \, m)$ our next task was to investigate how to efficiently calculate this value. The main problem encountered is with the b^e value becoming too large, causing the data to re-wrap producing invalid results. Initially we used the **Memory-efficient method** which did allow us to calculate the correct results without re-wrap, however it was rather slow. Thus, after further investigation we implemented the **Right-to-left binary method** which significantly improved the time of decrypting our data.

For our research we focused on implementing the RSA Encryption, however, we also felt that it was necessary to look into techniques used to crack RSA Encryption. Cracking an RSA encryption proved to be straightforward and well-known, albeit extremely difficult. The complexity of the task doesn't come from a complex code or algorithm standpoint, but from the sheer amount of computing power and time required to calculate the private key that is used to decrypt the message.

Attempting to factor the given **N** value, that being the product of two large prime numbers, and use that in conjunction with the given **public key** to solve for the decryption or **private key** is known as **brute-forcing.** This method of cracking RSA encryption is tedious and requires many years of computing time for even small keys to be cracked. Multi-threading can be utilized to run these brute force programs on multiple cores of a computer at a given time, each trying to factor *n*. Utilizing this technique, a 64-bit key is considered to be easily factored and cracked. Because of this, most practical applications use 128-bit and 256-bit keys for a much stronger encryption, while 1024-bit is used for an even higher levels of security. Methods used to brute force RSA encryption have been refined and optimized relentlessly by cryptanalysts, driving future advancements in encryption algorithms as well as forcing the use of ever larger keys and prime numbers to be used to stay ahead of new cracking methods.

In fact, new and incredibly fast and efficient methods of factoring numbers have been discovered, following the constant advancements of encryption and decryption methods in the last two decades. One of the fastest factoring methods, known as MPQS (Multiple Polynomial Quadratic Sieve) was able to factor a 256-bit *n* in only 35 minutes. The rising popularity of PC gaming and computational power of GPU's (Graphics Processing Units) has complicated this issue of easily factoring *n* further. With SIMD (Single Instruction Multiple Data) processing being run on graphics cards, *n* can be factored in a fraction of the time it would take a CPU running SISD (Single Instruction Single Data). This requires alternate methods to be employed to complicate the cracking process that aren't simply making keys larger and harder to factor.

Towards the end of implementing the RSA algorithm, we found that our implementation was susceptible to a couple of different attack methods. First off, due to the way we were initially padding the data during encryption, our implementation was highly susceptible to what is known as a **chosen plaintext attack**. This is when the cryptanalyst select arbitrary messages and has the algorithm encrypt them. In the worst case, the cryptanalyst can actually discover the private key through this method. The reason our implementation is so susceptible to this kind of attack is that we are not padding the data with random data to ensure it does not have a chance to look like plaintext once encrypted. A simple implementation to solve this problem is a cryptography standard known as **PKCS**. However, this type of padding is vulnerable to a type of attack known as an **Adaptive Chosen Ciphertext attack.** The solution to guard against this type of attack is **Optimal Asymmetric Encryption Padding**, or **OAEP**. While implementing this was outside of the scope of the project, what it does is hash the data with a hash generated from a random seed, then hash the seed using the hashed data. Then you store both the hashed seed, and the hashed data as the encrypted data. This means that the encrypted data looks different each time it is encrypted since it uses a random seed. However, for our project, we decided **OAEP** was outside of our scope since our project is focused on the actual RSA Algorithm for encryption and both OAEP and PKCS happen before and after RSA is run.

In conclusion, we discovered how to implement prime factorization and the RSA algorithm. We also discovered the ever increasing complexity of the issue of encryption and the race between encryption methods and cracking methods. Finally, we analyzed our implementation of the RSA algorithm to find shortcomings of our implementation, identified methods that take advantage of our implementation's weaknesses, and identified solutions to our implementation's shortcomings.