# CS Capstone: RSA - Cryptosystem Tool & Cognitive Analysis

George Wood

May 2018

## 1 My Capstone Experience

The process of the completion of my capstone, an RSA cryptosystem tool implemented in C++, has been long-term, confusing, and complicated. However, after all of the time and effort that has been poured into it, I can easily say that it has been the most fulfilling and high-quality software project that I have ever produced, independently or otherwise. In order to complete it, it was necessary to make use of my knowledge of object-oriented programming, filesystems, external libraries, and computer security, among many other aspects of my education at Truman.

Development of this project has spanned since roughly October of 2017, and has been quite time-intensive. Many setbacks and modifications were experienced with the specifics of the implementation, changes were made to the desired result of the project, and at one time there was the potential for it to be utilized in a research project. Despite the tumultuous nature of its development, I feel that the end result of these various changes is a notable accomplishment in programming, project management, and cybersecurity.

In addition to serving as the writeup for my computer science capstone, as part of the requirements of my cognitive science minor, a cognitive-science based analysis of RSA and why it would not be feasibly used prior to modern computers is provided in section 1.2 below.

### 1.1 Project Description

The program that I have produced is a fully-functioning RSA cryptosystem tool. RSA – short for Rivest-Shamir-Adleman, the last names of the scientists behind its creation – is a cryptosystem primarily used for secret sharing. It operates under the concept of public-key cryptography. Public-key cryptography makes use of both a public key and a private key, where messages encrypted with the public key can only be decrypted with the private key, and vice versa. In the specific context of RSA, the following values are utilized for the keys:

- The public key, generally notated as $e$. This key does not need to be kept confidential, and can often just be a small prime number, such as 3.
- The private key, generally notated as $d$. The secrecy of this key is crucial for maintaining the confidentiality and privacy of any and all information processed by RSA. It is generally an extremely large number, under FIPS guidelines either 2048-bit or 3072-bit.

- The modulus value used in internal calculations, generally notated as $n$. The size of $n$ in bits corresponds to the size of the private key $d$ in bits. RSA, unless non-conventionally implemented as a block cipher, cannot encrypt data that is larger than the size of $n$ in bits.
- The prime numbers used to determine the actual value of the modulus value $n$, generally notated as $p$ and $q$. These values are often extremely large, making the process of ensuring that they are prime a notable challenge. Although RSA technically still functions if $p$ and $q$ are not prime, the renowned security of RSA relies of the difficulty of the factorization problem, which is weakened substantially if these values are not prime.

The capabilities and features of my implementation are as follows:

- FIPS Compliance (with one exception)
  - Every aspect, with one exception, of the program follows the specifications set by NIST (National Institute of Standards and Technology) in their FIPS (Federal Information Processing Standards) publications. These collections of standards set the requirements and guidelines of software that is to be used in a governmental or otherwise security-prioritizing context. They provide formal – albeit sometimes unclear – statements of implementation features that have been determined to provide the degree of security and confidence that would be required in high-security contexts.
    * The only aspect of the entirety of my code that is not compliant with FIPS standards lies within the random number generator that I created (discussed further later). True FIPS-compliance in random number generation requires an external noise source, such as the famously reported usage of walls of lava lamps with cameras pointed at them to generate random bits. Inclusion of this requirement in my implementation not only would have added substantial costs of both time and money, but more importantly seemed unfeasible in the first place given the scope of the project.
- Encryption, Decryption, Signing & Authentication
  - The program can perform the four standard RSA operations of encryption, decryption, signing, and authentication, provided a key pair and input data. All input and output data is retrieved and stored in text files accessible by the program, which must contain data encoded in hexadecimal. All of these operations will not be performed if no key data is available. Explanation of these operations is as follows:
    * Encryption/Decryption: A message encrypted with the public key can only be decrypted with the private key. This is useful if, for example, an individual wants anybody to be able to send them a message, but only wants themselves to be able to read it. Since anyone can have access to the public key, anyone can encrypt with it. However, only the original individual, the holder of the private key, will be able to decrypt and obtain the original message. The equations below are utilized for these operations, with equation 1 being for encryption, and equation 2 being for decryption. $m$ represents the original message, and $c$ represents the encrypted message:

      $$[1] c = m^e \ mod \ n$$

      $$[2] m = c^d \ mod \ n$$

* Signing/Authentication: A message encrypted with the private key (signing) can only be decrypted with the public key (authentication). This is useful if, for example, an organization wants to send a message to all interested parties, while those parties can be assured that the message came from the party that the sender claims to represent. If the message signed with the private key can be properly authenticated with the public key, the recipients of the message can be confident the message is authentic. This is because only the original party has access to the private key, and the public key would only authenticate the message properly if it was signed with the genuine private key. The equations below are utilized for these operations, with equation 1 being for signing, and equation 2 being for authentication. $m$ represents the original/authenticated message, and $c$ represents the signed message:

  [3] $c = m^d \ mod \ n$

  [4] $m = c^e \ mod \ n$

- Chinese Remainder Theorem Option
  - to reduce computational demands and execution time. Although it performs the same function as the standard decryption/signing functions listed above, it is capable of doing so with fewer internal operations and/or more minor computational complexity. The equation for this is as follows:

    Both the decryption and signing operations have the option to utilize the Chinese Remainder Theorem, or CRT, as an alternative mechanism that can increase computational efficiency. It performs the same actions in effect, but can do so with fewer internal operations and/or a lesser degree of computational complexity. Using this scheme, intermediate values determined using $p$ and $q$ in addition to the modular inverse of $q \ mod \ p$ ($qInv$) are utilized. These values are determined as follows:

    $dP = (1/e) \ mod \ (p-1)$

    $dQ = (1/e) \ mod \ (q-1)$

    $qInv = (1/q) \ mod \ p$

    These values can then be used to compute the output message without such costly operations as $modn$. Using decryption as an example, this is accomplished as shown below. $m_1$ and $m_2$ are intermediate portions of the decrypted/authenticated message, and $h$ is another intermediate value used.

    $m_1 = c^d P \ mod \ p$

    $m_2 = c^d Q \ mod \ q$

    $h = qInv(m_1 - m_2) \ mod \ p$

    $m = m_2 + (h)(q)$

Signing with CRT can be accomplished in the same manner, simply swapping the positions of $m$ and $c$.

- Key Generation
  - The key generation implementation is easily the most complex portion of the software. Whereas the bitlength of the modulus value $n$ can technically be smaller than that specified by the type of RSA (e.g. 2048 bits for RSA-2048), and the bitlength of the private key $d$ can technically be smaller than that of $n$, this implementation *guarantees* that the bitlength of both $d$ and $n$ will be exactly that specified by the type of RSA. Additionally, despite being absolutely massive numbers, the primes $p$ and $q$ are both guaranteed to be provably prime. A sieve procedure in combination with an iterative testing procedure will not allow a non-prime value to be output for these values, as can be seen during the key generation process via the displaying of a number of tests.
- Key Saving/Loading & Key Size Options
  - The cipher is generated anew each time the program starts, meaning it starts with no key information. This poses a problem for situations where some data is encrypted at one time, but must be decrypted later. To remedy this, any keys can be saved and restored in a later instance of the program. All saved keys can also be viewed without immediately changing the current keys. Additionally, the two FIPS-approved key sizes, 2048 and 3072-bit, are available as selectable options. Only keys that will work with the currently selected key size are displayed as loadable options.
- Entirely Hand-Written Code (with one non-standard library)
  - Every portion of the code, except for one non-standard C++11 library, was entirely written by me. This includes (naturally) the RSA cipher itself, in addition to the user interface code, and the SHA hashing class and random number generator used for key generation.
    - The SHA hashing class, SHAHash, is capable of both SHA-224 (specified by FIPS to be used with RSA-2048) and SHA-256 (specified by FIPS to be used with RSA-3072). Both of these versions of SHA are implemented in the same class with the version to be used specified in the constructor of the class, which is specified based on the type of RSA being used.
    - The random number generation class, RandGen, is capable of generating random streams of an arbitrary and specified number of bits.
  - The only non-standard library utilized was GMP, the GNU Multiprecision Arithmetic library. GMP allows for arithmetic to be performed on numbers of an arbitrary size, surpassing the 32 or 64-bit limitations present on most computer architectures. The complexity of such functionality is monumental, and would constitute much more than a capstone project in itself. So, in order to make the completion of my project more feasible, I elected to use this well-established and highly-regarded library.

### 1.2 Cognitive Analysis

One of the most notable aspects of RSA, in my opinion, is the absolute unfeasibility of performing it by hand. Prior to the advent of modern computing, such data processing would require computations of a level of complexity that, even if one were capable of and willing to take the massive amount of time to perform it, would render the system nearly impossible to use (of course, prior to this technology, such degrees of encryption very likely may not have been necessary).

- Rule of Seven Plus or Minus Two
- Ability to Count on Sight
- Organizational/Mathematical constraints

## 2 Project Environment

The environment of my project was fairly straightforward. Nearly all of the work was completed at my personal workstation in my home, with small exceptions of information gathering being performed on my laptop the Pickler Memorial Library. The social environment was minimal, as all of the work was performed by me. There were periodic meetings with Dr. Jaiswal to provide updates of project status and exchange information on potential additional features. The development environment largely utilized an agile process. Changes and additions were made in small increments, and focus would remain on each individual issue until they were resolved, rather than moving on and returning later.

## 3 Other Involved Parties

I worked with others to a rather minimal extent. Beyond my interactions with Dr. Jaiswal, the entirety of the project was planned and executed by me. In order to ensure the software was usable by individuals unfamiliar with RSA, I had several friends use the program and inform me of what was straightforward and what features may need further clarification.

## 4 Hardware/Software Platform

Although I did perform some information gathering and research on my laptop, the entirety of actual programming was performed on my desktop computer. In terms of hardware, it far exceeded any and all requirements for this project. It has an Intel i7 4790k processor, 8GB of DDR3 RAM, and an Asus Maximus VI motherboard. The operating system it runs on is Windows 7 Home Premium 64-bit, but in order to ensure fewer compatibility issues with UNIX-like systems, I did not use Visual Studio or other forms of native Windows C++. Instead, I utilized MinGW, a development environment that simulates GNU on Windows platforms and allowed me to compile my code with gcc. To actually write the code, I utilized the Atom text editor with a gcc compiler plugin. In order to keep track of the project as it developed, I made use of git, often through the interface provided by application GitKraken.

## 5 Communication Skills

In order to communicate with my supervisor, Dr. Jaiswal, I relied primarily on email and in-person meetings. Minimal code was shared over our correspondance, but general topics and specific implementation issues were discussed in a general sense. The ability to discuss code in a concise and descriptive manner that I have developed over my time as a student proved to be extremely useful. When working with numbers as large as are utilized in RSA and the procedures to generate them, it quickly became clear how difficult speaking of the code relevant to them would be.

In order to find people to test my software, I largely relied on word of mouth and phone calls.

## 6 Useful CS Courses & Topics

- **CS 250 - Systems Programming.** The knowledge that CS 250 imparted to me about bitwise operations and lower-level programming proved invaluable in the creation of my SHA class.
- **CS 260 - Object-Oriented Programming and Design.** Principles of object-oriented programming, especially that of encapsulation, facilitated efficient and effective interactions between the classes I designed.
- **CS 370 - Software Engineering.** Without my knowledge and usage of Git, which I first became familiar with in this course, there were a number of times were I may have suffered data loss. Equally importantly, knowing how to manage such large and complex code as was necessary for this project would have proven immeasurably more difficult without the techniques taught in this course.
- **CS 390 - Operating Systems.** CS 390 showed me an area of computer science that I was largely unaware of prior to taking the course. Without understanding the specific differences between Windows and Linux, for example, making use of MinGW and the GMP library would been a much more confusing process.
- **CS 455 - Computer Security Fundamentals.** Without a doubt, CS 455 provided the foundation of this project for me. I was completely unaware of how interesting I would find the field of cybersecurity as a whole, and had no knowledge of the specifics of encryption systems such as RSA. Upon hearing of RSA in this course, I immediately knew that I would be interested in a capstone project involving it.

## 7 Useful Non-CS Courses & Topics

- **MATH 263 - Analytic Geometry & Calculus II, & MATH 285 - Matrix Algebra.** Without the upper-level mathematical concepts taught in these courses, the idea of working with numbers larger than a quattuorducentillion would be absolutely unfathomable to me.

# 8 Lessons Learned

1. **Know what you're going to make before you make it.** Over the course of this program's development, a considerable number of changes were made to the intended end-product. Due to this, time spent researching was wasted, well-designed code was scrapped, and conflicting information gathered caused confusion. Although changes to the specific intentions and goals of a program are often inevitable, a more formalized plan of my end goals being prepared from the beginning would have saved a lot of expended time and effort.
2. **Test portions of large programs in small parts.** This was a principle I learned in CS 370, but as this project has easily been the largest programming project I have ever completed, this lesson was quite strongly reinforced for me. The nested nature of many of my functions, in addition to the complexity of the mathematics being performed and the sometimes unclear FIPS specifications, made unit testing and other function-specific testing difficult. Although I did perform testing to a considerable extent, additional time spent planning more complete tests would have, despite taking more time initially, saved me time in the long run.
3. **Coding in accordance with external specifications can be difficult, to say the least.** Writing my program in alignment with FIPS specifications was an arduous process. Sometimes, the meaning of a particular statement or mathematical expression was incredibly unclear. The symbols of variables and operators were sometimes inconsistent, occasionally within the same publication and very often between publications. Although I am quite sure that – and as I experienced in CS 370 – programming according to commercial, client requirements is quite different from doing so according to federal standards, the lesson is the same: Producing something according to needs sourced from a third party can lead to countless opportunities for confusion, misinterpretation, and improper results.

# 9 Problems & Solutions

# 10 Things I Wish I had Known

- **GMP, a library designed for C, does in fact have an extended C++ library.** In my initial attempts to create this program, I was of the belief that, although incredibly powerful, the GMP library only provided types and structs that had access to the multiprecision functions. This caused sections of code that would ordinarily take only one line require numerous, generally around four, but occasionally higher. The way that access to stored variables and functions was structured made the code much less easily readable, leading to my own code occasionally being difficult for even me to understand, despite my best efforts to comment and document dutifully.

  After struggling greatly with debugging this code, I decided to look more deeply into the GMP documentation, and discovered an extended library that incorporates all of the multiprecision functionality into C++ classes. This easily halved the number of lines in my code, and even more importantly made mathematical expressions appear much more natural and readable.
- **Creation of GUIs is incredibly complex.** For some time, I had planned for this program to implement an aesthetically pleasing and easy to use GUI. At separate times, I had

attempted to use both WxWidgets and Qt to create a GUI, and both times quickly found myself in over my head. I persisted, however, and all I ended up achieving was confusing myself and modifying my code in detrimental ways – thanks to Git, no permanent damage was done. GUI programming is an art and a craft in itself, and had I known the immense difficulty associated with it beforehand, I would have been aware that creating a GUI, at least in C++, would be a project in itself and likely draw attention away from the true intention of my program.

## 11 Results: Anticipated vs. Actual

After all is said and done, I am very pleased with the end result of this project. The cryptosystem tool works perfectly in terms of its intended operations, and exhibits no errors that I or my testers have been able to find (while I am aware that testing can only show the presence of bugs, rather than a lack of them). My initial hopes for the project, looking back, were quite grandiose, and I now know that they could easily be described as overly ambitious.

Initially, I had envisioned a cryptosystem tool with a fully functioning GUI that could demonstrate encryption/decryption or signing/authentication between two separate ciphers, including showing what would occur if incorrect keys were utilized. I also had ideas of being able to demonstrate this via two instances of the program connected over a network. However, as soon as the difficulty of creating a GUI became apparent, devising an approach to this that would be easily-interpretable while still being constrained to the command line seemed unlikely. Additionally, I severely underestimated how complicated the procedures outlined in the FIPS publications would be. I had pictured it as being a number of additions to the simple series of equation presented in CS 455, while the reality of it was a volume of additional steps that were much more difficult for me to understand than the core equations. All of this led to me cancelling those plans, and focusing on a command-line based, strong-security cryptosystem.

While I one day would very much enjoy being able to return to this project and add some of the functionality that I ended up having to scrap, as of right now I strongly feel that the product of this capstone is an appropriate culmination of my time as a computer science student at Truman. It required hours of research and coding, success and failure, pride and frustration, and most of all testing and improving. I certainly have never faced such a difficult programming before, and believe I have tackled it to an extent that I am very proud to call it my capstone.