Final Report
Crypto Playground: A Computer Security Learning Tool

Quinton Teas, B.S. Major in Computer Science
Philip Passantino, B.S. Major in Computer Science
Egan Dunning, B.S. Major in Computer Science
Advised by Dr. David A. Wolff

Pacific Lutheran University

Spring 2017

May 25, 2017

Contents

1 Introduction **4**

2 Functional Requirements **4**

3 Non-functional Requirements **4**

4 Design **5**
- 4.1 Overview 5
- 4.2 Cryptography Game 5
- 4.3 Hash Breaking 7
- 4.4 Factorization 10
- 4.5 Graphing 12
- 4.6 Website 14

5 Implementation **14**
- 5.1 Agile Development 14
- 5.2 Tools 15
- 5.3 Issues 15

6 Future Work **16**
- 6.1 Regression Line Prediction 16
- 6.2 Encrypt Quote File 16
- 6.3 Finish Implementation of Quadratic Sieve 17

Glossary **19**

List of Figures

1 UML Diagram for the project structure. 6
2 SSD for the cryptogram game. 7
3 Hash function classes overview 8
4 Difference in hash cracking times using 1, 4, and 8 threads. . . 9
5 UML class diagram for the factoring feature 10
6 SSD for graphing hash cracking results. 13
7 Screenshot of the Qt Creator Window Builder. 15

Abstract

We created a program that teaches its users computer security concepts. The program, a cross platform desktop app built in Qt, explores security technologies. Users may play a fun cryptogram game, explore password security with a hash cracking tool, and factor miniature RSA numbers. Using this application, we can show our users circumstances where encryption is secure and circumstances where encryption is not secure. We can also show our users the difference between strong and weak passwords.

1 Introduction

We began this journey nine months ago when we decided to become a team and design a desktop application. After nine months, we are proud to present our educational desktop application: Cryptography Playground. There are three parts that make up Cryptography Playground: the cryptography game, the hash breaking algorithms, and the factorization methods.

2 Functional Requirements

Cryptogram game Our first requirement is a cryptography computer game. The cryptography game introduces users to the complex idea of encryption using the concept of a simple cipher.

Hash breaking Our second requirement is reversing Hash Functions. Our motivation is to test the strength of passwords. The hash breaking algorithms we chose to implement were brute force and dictionary attack.

Integer factorization Our third requirement is integer factorization. Since RSA encryption can be broken by factoring a large number, we are interested in finding quick and efficient ways of factoring integers. We chose to implement two factoring algorithms: trial division and quadratic sieve.

Multithreading Our final requirement is multithreading. Since hash breaking and integer factorization are computationally intensive, we wanted to speed up these computations by distributing the workload to more than one thread. This allows machines with multiple CPU cores to be able to crack hashes and factor numbers faster.

3 Non-functional Requirements

Educational We want this application to give any user easy access to learn about security methods that exist today. Our cryptogram game was added as an introduction to ciphers and security. Users then have the option to learn about more complex topics, such as hash breaking. With the hash breaking features of our program a user who has no knowledge of hashes can see how hashing works and when hashing is effective at hiding data. Our goal was for our users to gain information about modern security methods first hand within the first ten minutes of them opening up the application.

Cross-platform In the interest of sharing our application, we did not want to limit our program to only one platform, so we made it a goal to only use technologies that were cross platform. Being able to supply our program to all three major operating systems is a great way to include most users' machines given their choice in operating system.

4 Design

4.1 Overview

Crypto Playground has three tabs which separate the three main parts of the program.

Figure 1 is the UML diagram for Cryptography Playground; dotted lines show dependency and solid lines show association. Our overall structure has the `MainWindow`, which is the controller, and then our three features are all implemented in the `CryptoGame`, `Crack`, `Hash`, and `Factor` classes. Other classes such as `LabelArray`, `ButtonArray`, and `GraphWindow` are all user interface classes we've made to assist our graphical user interface with displaying graphing information and programming more complicated user interface designs than the Qt layout designer can perform. Not shown is the user interface class that is generated from the Qt layout designer.

4.2 Cryptography Game

We needed a way to teach our users about the complex ideas of encryption and hashing that would not overwhelm them. On the recommendation of Dr. Wolff, we decided to use a computer game of a cryptogram to achieve this. We made it our goal to design this cryptography game in such a way as to have the fun characteristics of the pencil and paper version while simultaneously adding new features for the players' convenience.

We kept two main features from the pencil and paper version: multiple lines and a hint for the player. These features are included in the cryptogram book "Cryptograms to Keep You Sharp," which we used as inspiration for the

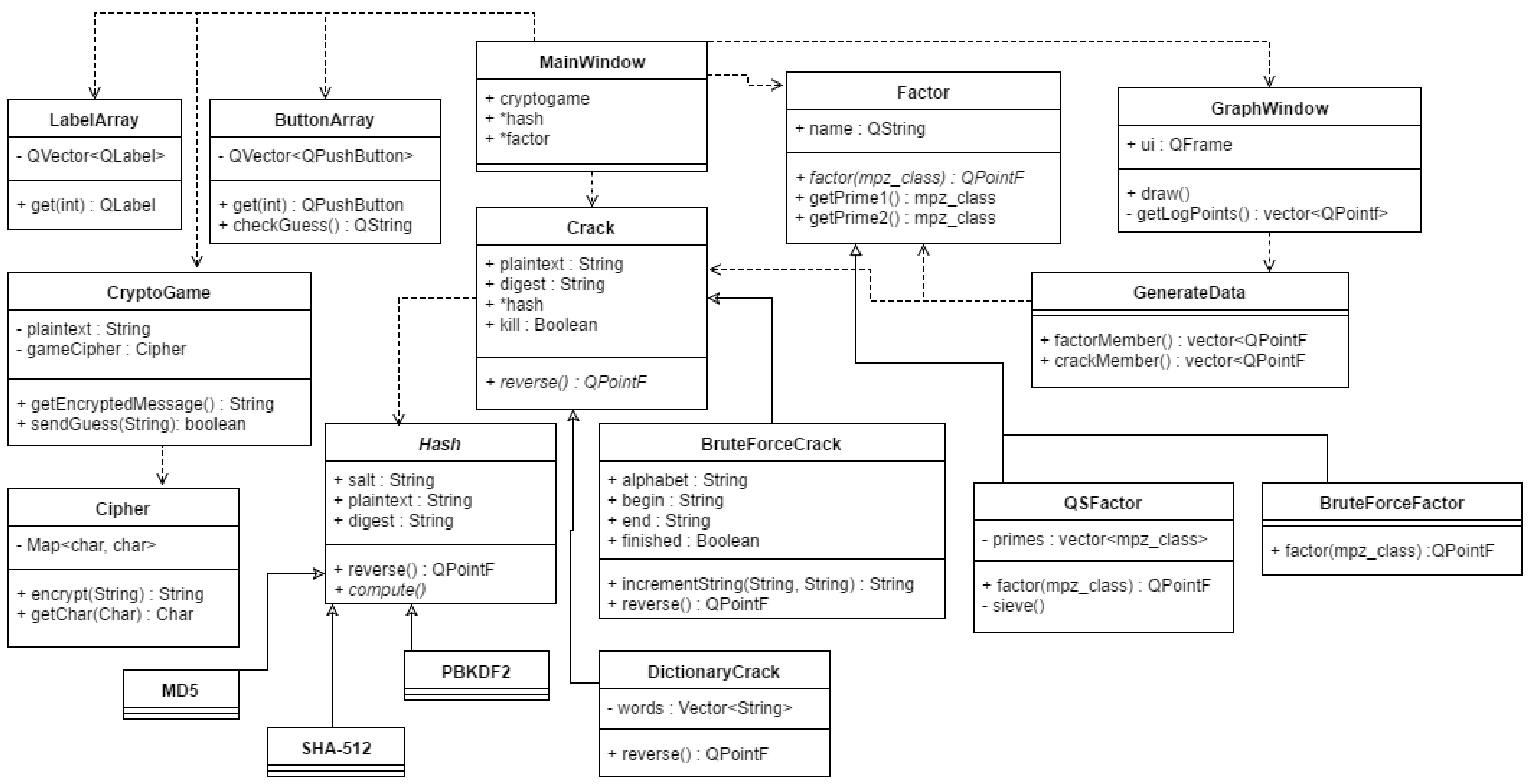


Figure 1: UML Diagram for the project structure.

game's phrases and the game's hints.[1] The game's encrypted phrase and the buttons beneath it span multiple lines. This makes the game window appear uncluttered for the player. Hints are common on the pencil and paper versions of cryptograms, so we made sure the player can click a button to get a hint for the first letter of the phrase.

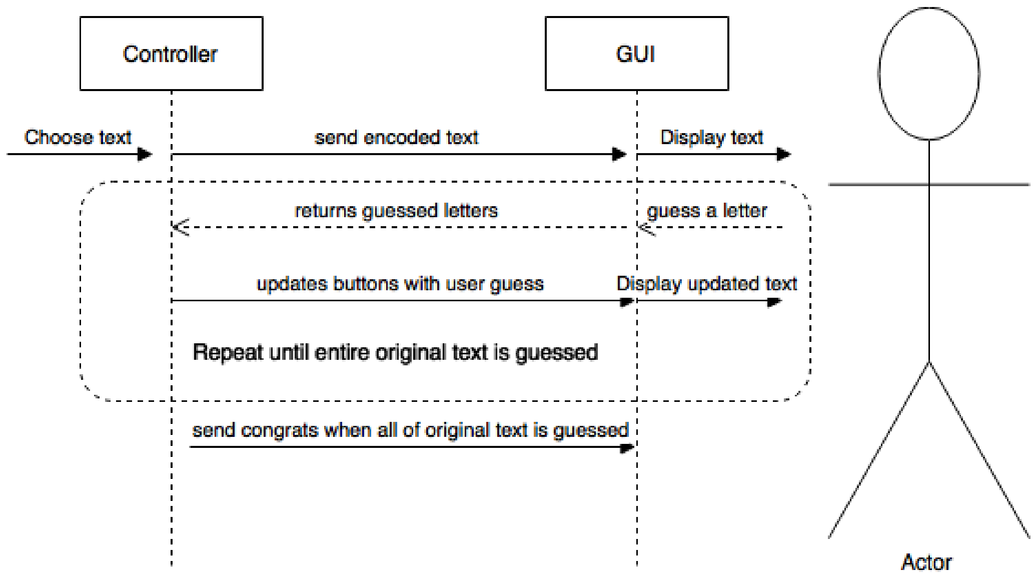

Controller
GUI
Choose text
send encoded text
Display text
returns guessed letters
guess a letter
updates buttons with user guess
Display updated text
Repeat until entire original text is guessed
send congrats when all of original text is guessed
Actor

Figure 2: SSD for the cryptogram game.

Due to the nature of computer games, we added two features not found in the pencil and paper version of cryptograms: automatic letter marking and automatic victory checking. When the player makes a guess on a letter, the game will automatically mark all letters in the phrase that have the same encrypted letter. Automatic victory checking means the game will check the guessed phrase after every user input. This allows players to know if they are correct immediately upon guessing the correct phrase.

4.3 Hash Breaking

In the hashing section of our program, our primary goal was to showcase different hashing methods and how they work. We included a graphing method to compare the hash breaking times of our separate methods. We also have a hash breaking method that can attempt to reverse the result of hash functions.

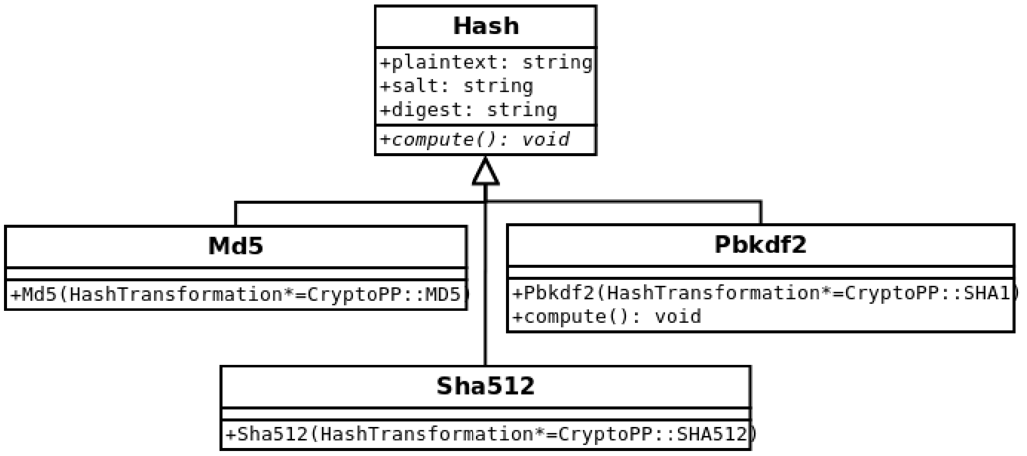

Hash
+plaintext: string
+salt: string
+digest: string
+compute(): void
Md5
+Md5(HashTransformation*=CryptoPP::MD5)
Pbkdf2
+Pbkdf2(HashTransformation*=CryptoPP::SHA1)
+compute(): void
Sha512
+Sha512(HashTransformation*=CryptoPP::SHA512)

Figure 3: Hash function classes overview

We had to import our hashing function implementations from Crypto++, yet we did not want to reuse large amounts of code when using these methods that were implemented already. So we created the `Hash` class which we made super class to three of the hashing functions that are implemented in Crypto++. Instead of calling from Crypto++ we instead can now declare a hash variable and instantiate any of the three hashing implementations. This also gives us easy access to add more hashing function implementations, in which case we would add another subclass to `Hash` for the new implementation and implement `compute()` which all subclasses of `Hash` must have implemented.

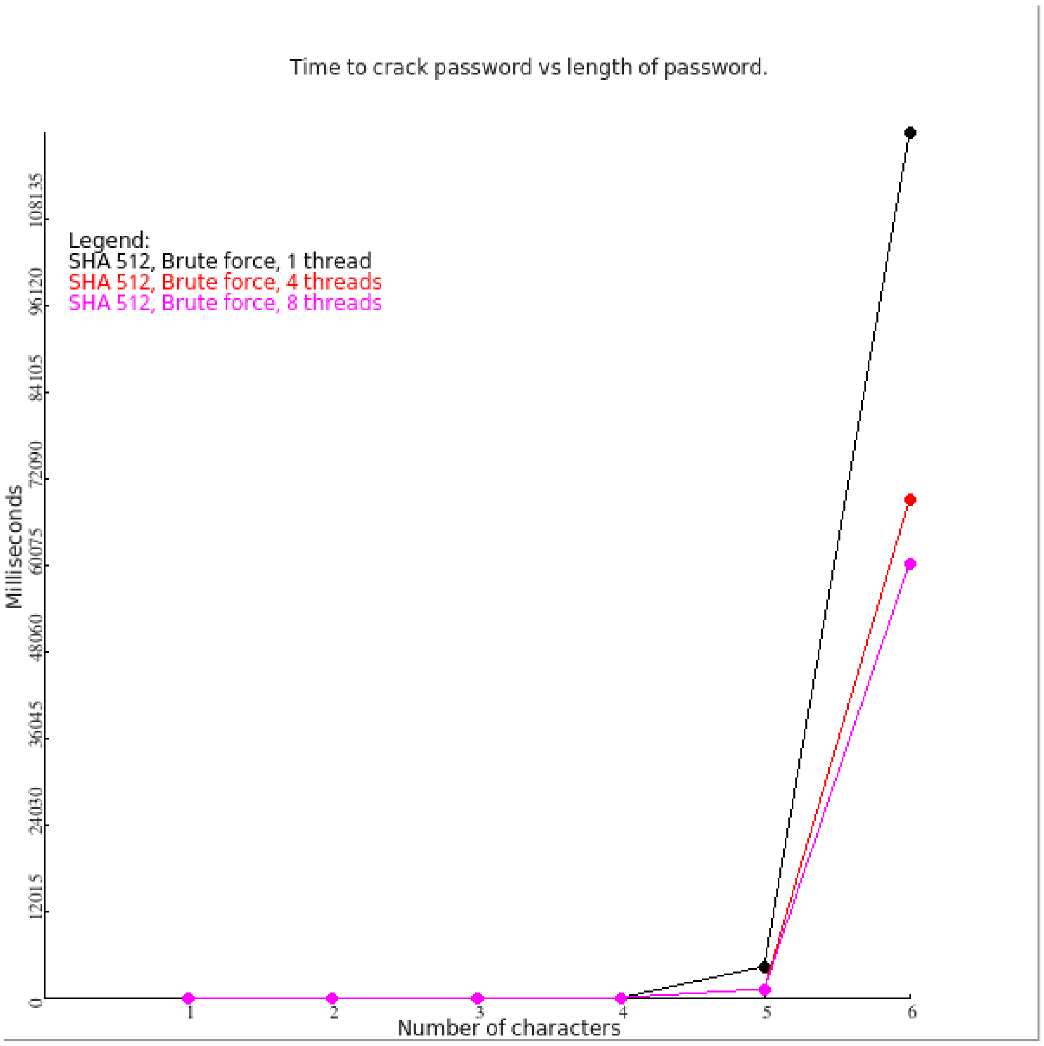


Figure 4: Difference in hash cracking times using 1, 4, and 8 threads.

For our brute force hash cracking function, we implemented it so that multiple threads can be used to reverse a hash. The optimal amount of threads, determined by Qt's `QThread` class, is selected as default. The user can change the number of threads used to see the performance benefit from increasing the number of threads.

4.4 Factorization

We are interested in factoring large integers because RSA encryption can be broken by factoring a large integer. The RSA public key is a pair of integers, the RSA modulus and the public exponent. The RSA modulus, or RSA number, is a product of two large primes. NIST recommends a RSA modulus size of 2048 bits or more [2].

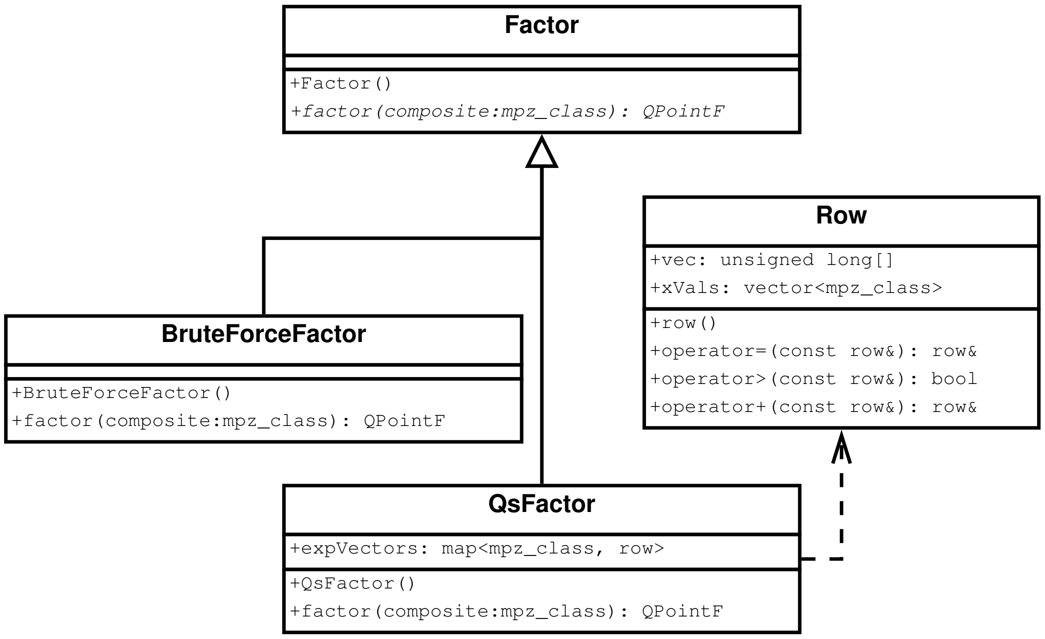

Factor
+Factor()
+factor(composite:mpz_class): QPointF
Row
+vec: unsigned long[]
+xVals: vector<mpz_class>
+row()
+operator=(const row&): row&
+operator>(const row&): bool
+operator+(const row&): row&
BruteForceFactor
+BruteForceFactor()
+factor(composite:mpz_class): QPointF
QsFactor
+expVectors: map<mpz_class, row>
+QsFactor()
+factor(composite:mpz_class): QPointF

Figure 5: UML class diagram for the factoring feature

The design for the factorization feature is similar to the hash cracking design. There is a parent class, `Factor`, and child classes that inherit from factor. Child classes override the `factor()` method, and implement a factoring algorithm. We implemented two factoring algorithms: brute force and quadratic sieve.

We used the trial division algorithm in the brute force implementation. Trial division searches for factors of N by dividing N by integers less than or equal to the square root of N. If a number evenly divides N, it is a factor. In the worst case, trial division takes $\sqrt{N}$ steps. If N has n decimal digits, the worst case efficiency is $10^{n/2}$. This means trial division has exponential time complexity in the length in digits.

The quadratic sieve algorithm is much more complex, but can factor large numbers faster than trial division. To factor a product of two primes n, the basic idea is to find integers a, b that satisfy:

1. $a^2 \equiv b^2 \mod n$
2. $a \not\equiv \pm b \mod n$

Integers u, v are said to be equivalent mod n if $u - v$ is a multiple of n. We denote this as $u \equiv v \mod n$. Once a, b are found, then the prime factors of n are $\gcd(a + b, n)$ and $\gcd(a - b, n)$. This idea is the basis for many modern factoring algorithms [3].

The quadratic sieve is a method for finding a, b that satisfy the above properties. Look at values of the polynomial $Q(x) = x^2 - n$ for integers $x > \sqrt{n}$. If $Q(x_1) \times Q(x_2) \times \ldots \times Q(x_k)$ is a square, then

$$Q(x_1) \times Q(x_2) \times \ldots \times Q(x_k) \equiv x_1^2 \times x_2^2 \times \ldots \times x_k^2 \mod n.$$

To find a product of polynomials equal to a square, we generate exponent vectors for each polynomial. Let a be an integer that completely factors up to prime numbers less than B. We define the exponent vector for a in the following way.

$$a = \prod_{i=1}^{\pi(B)} p_i^{v_i}$$

$$\mathbf{v}_a = (v_1, v_2, v_3, \ldots, v_k)$$

The first equality shows the prime factorization for a. The function $\pi(B)$ is the prime counting function, its output is the number of primes less than B. p_i is the ith prime, v_i is the exponent for the ith prime, and is the ith entry in the exponent vector for a.

If the exponent vector for a has only even entries, then a is square. To verify this, suppose $\mathbf{v}_a$ has only even entries. Then

$$a = \prod_{i=1}^{\pi(B)} p_i^{2v_i} = \left(\prod_{i=1}^{\pi(B)} p_i^{v_i}\right)^2 \text{. So } a \text{ is square.}$$

Furthermore, if the exponent vector for a is equivalent to the zero vector mod 2, then a is square. With this fact, we know how to identify square numbers. To represent the product of integers with an exponent vector, we only need to add the exponent vectors of the factors. If we multiply integers a, b, then $\mathbf{v}_{ab} = \mathbf{v}_a + \mathbf{v}_b$. Now we have a problem that we can solve using Gaussian elimination. We can put exponent vectors for values of $Q(x)$ into a matrix with entries in the set $\{0{,}1\}$. Then use Gaussian elimination to row reduce. During the Gaussian elimination step, the only operation we need to use is addition mod 2, denoted $+_2$. We only need to use addition mod 2 because $1 +_2 1 = 0$, meaning we can reduce rows using only $+_2$. Each vector addition operation represents multiplying the corresponding integers. Now all zero rows correspond to a square. This provides us with a congruence of squares mod n, so we can compute the factors of n using the Euclidean algorithm as stated above[4].

In our implementation, exponent vectors are stored as a `long int` array. Each bit in a `long` represents an entry of an exponent vector reduced mod 2. Each exponent vector is stored in the `Row` class, which represents a row in the matrix of exponent vectors. The `Row` class has two attributes, the exponent vector for $Q(x_1) \times \ldots \times Q(x_k)$ and a `std::vector` containing $x_1, \ldots, x_k$. When two rows r_1, r_2 are added, the exponent vectors are added mod 2, and the x-values for r_1 and r_2 are put into the same `std::vector`. Addition mod 2 is done using bitwise XOR for each `long` in the exponent vector.

4.5 Graphing

For the Hashing and Factorization features, the user can use the included graphing feature. This graph lets users compare the amount of time our algorithms would take to crack the hash or factor the given number. We included a logarithmic scale option for the graph, which makes it easier for the user to observe a graph with a large range of times.

We used separate classes for data generation and graphing. `GenerateData` handles data generation for both factoring and hash cracking. For factoring data generation, the `composites(length, n)` method is called. The `length` parameter is the length in digits of the first composite number to be included in the data set. `n` is the number of composites to include in the data set. The `composites` method generates `n` composites, starting with `length` digits, and increasing by one digit. Each composite is a product of two primes. To generate a product of two primes with x digits, pick primes with y and z digits such that $x = y + z + 1$. To generate a random prime number with y digits, randomly generate a string of y digits and ensure that the last digit is odd. Then, use trial division to check if the number is prime. Trial division is slow, so generating very long prime numbers is impractical. The benefit of using trial division rather than a probabilistic primality test is that we can be absolutely sure that each prime number generated is prime. After `composites` is finished generating numbers, the `factor` method is called, which takes a `std::vector` of composites and a `Factor` object as parameters. The `factor` method records the time taken to factor each composite. When finished, `factor` outputs a list of point objects, containing the number of digits of the number that was factored, and the time taken to factor that number.

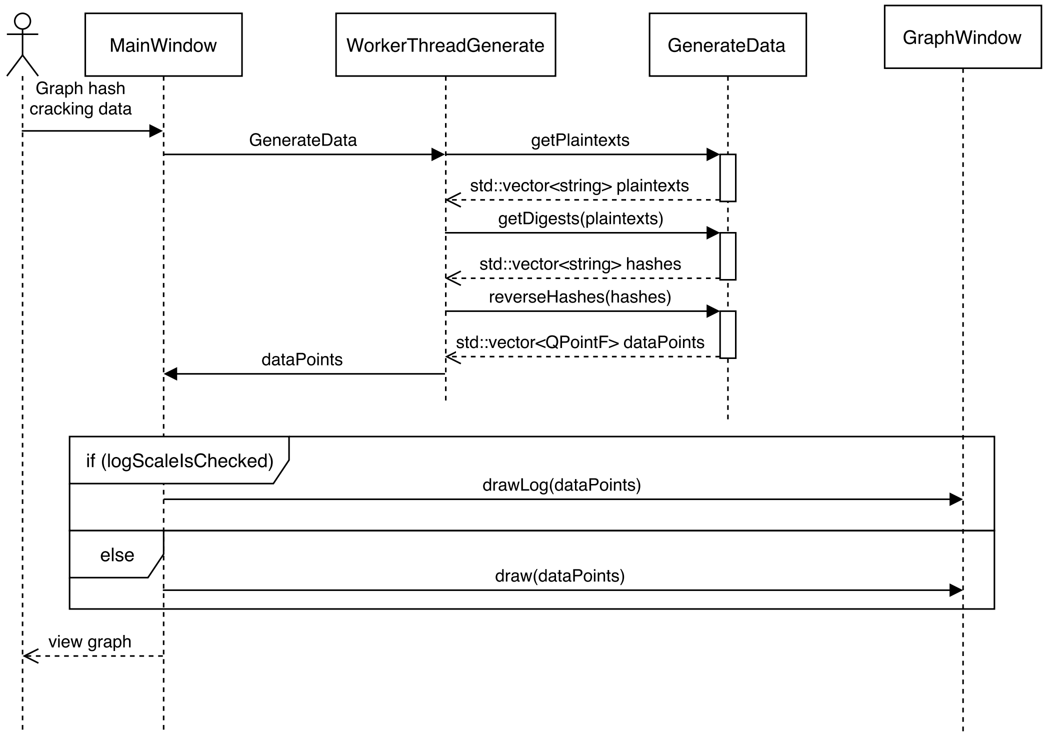


Figure 6: SSD for graphing hash cracking results.

Generating data for hash cracking is similar, first a list of plaintexts are generated, these plaintexts are hashed, then cracked. Again, the output is a list of points, where each point contains the length of a plaintext and the time taken to crack the corresponding hash.

Once the data points are generated, the points can be plotted by the `GraphWindow` class, which handles plotting, log scale transformation, axis drawing, legend drawing and title drawing. More points can be overlaid on the same axes, the color for each line graph is picked from a list of colors in order to distinguish between two data sets.

4.6 Website

To get a detailed description on how our program works we wanted to provided a resource for a user to obtain that information without it clogging the program's interface. As a group we decided it would be best to put off all of this information in a website so it would not clutter the program. The website is a static website that was built from the Express[5] framework using Node.js[6], and is hosted on Heroku[7]. Aside from the content in the webpages, and styling using CSS, the most technical aspect of the site is the routing which we handled through the index.js file.

5 Implementation

5.1 Agile Development

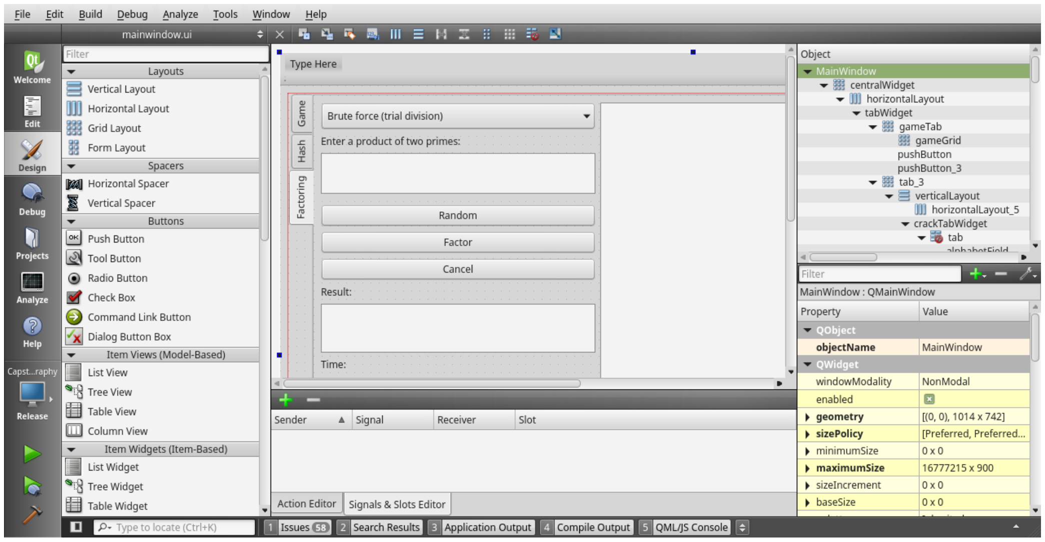
We used the Agile workflow to develop this program. Using Agile methodology, we were able to react and adapt to changes in our functional requirements without a fundamental redesign. For example, in our initial requirements we planned on using OpenCL to speed up hash breaking and factoring. However, we removed this requirement due to the steep learning curve for OpenCL. We adapted to this change, and decided to use multithreading to improve performance instead.

In the Agile methodology, development is done iteratively. During each iteration, or sprint, the developers go through the planning and implementation phases of software design. Every iteration should yield working software. Our sprints were two weeks long. Another aspect of Agile software development are stories. A story can be a software feature or other requirement.

Specifically, we followed the Scrum framework. This meant we held regular stand-up meetings during each sprint where each team member discussed progress since the last stand up, current task, and any limiting factors. At the beginning of each sprint, we decided which stories to work on over the next two weeks. At the end of each sprint, we demoed our progress and reviewed both completed stories and stories that we were unable to finish during the sprint.

5.2 Tools

We used many tools that we had little experince with to complete this project. Our code is written in C++ 11 and compiled with gcc. For version control, we used git, and hosted our repository on Github. Qt provided cross-platform GUI and multithreading libraries, and the qmake build automation tool[8]. We used the QtCreator IDE for editing code and the GUI. For big integer handling, we used the GMP library[9]. We used implementations of Hash Functions provided in the Crypto++ library[10].



Property	Value
QObject	
objectName	MainWindow
QWidget	
windowModality	NonModal
enabled	☒
geometry	[(0, 0), 1014 x 742]
sizePolicy	[Preferred, Preferred...
minimumSize	0 x 0
maximumSize	16777215 x 900
sizeIncrement	0 x 0
baseSize	0 x 0

Figure 7: Screenshot of the Qt Creator Window Builder.

5.3 Issues

Our team overcame some obstacles while we worked on this project. The first and most difficult obstacle we tackled was the problem of installing the GMP and Crypto++ libraries on the Windows and Mac machines. Without these libraries installed, the Windows and Mac machines could not run the project. This caused us several months of frustration as we tried and failed to install the libraries in the proper directories. At the end of January we decided continue developing the project using only the Linux Mint operating system. The Windows and Mac machines used the Virtual Box software to run Linux Mint. Switching to Linux solved all of our library problems, but it also indefinitely postponed our goal of creating a cross-platform application.

We had another problem we needed to overcome: our lack of meeting times. During fall semester we held our weekly stand up meeting over the Slack messaging service and met once every other week Sunday afternoon. This scheduling led to a slow development process and poor communication over Slack. We decided for spring semester that we needed to meet in person to work on the project and write code in the same room. We chose to meet three times every week on Tuesday, Wednesday, and Thursday. This scheduling, along with switching to using Linux, improved our development process tremendously.

6 Future Work

6.1 Regression Line Prediction

One of our goals was to use a regression line to predict how long it would take to factor very large numbers and crack long passwords. This feature would make the program more useful, since users would be able to get a better idea of how time consuming factoring and hash breaking can be.

For example, if a user wants to know how long it would take to factor a 2048 bit RSA number using trial division, the user could look at a regression line to predict how long this operation would take. This is preferable to letting the program run for years.

We did not have the time to implement this feature, but our design will be able to accommodate a regression line option for both factoring and hash breaking.

6.2 Encrypt Quote File

Currently, it is possible for our users to view all of the answers to the quotes if they look through the application data. Since our application is a security learning tool it would be good practice to secure our list of quotes from being seen or accessed.

6.3 Finish Implementation of Quadratic Sieve

The current implementation of the quadratic sieve algorithm is not fully functional. There are bugs in the linear algebra step that prevent the algorithm from finding the prime factors. Our implementation works for small numbers, which do not display the speed of the algorithm. The quadratic sieve is suited for factoring integers that are at least 20 digits long.

Our goal is to be able to compare the running time for brute force and quadratic sieve. Our prediction is brute force will outperform quadratic sieve for numbers with fewer than 18 digits. Brute force is very fast for small numbers, since the algorithm is simple and requires little to no initialization.

References

[1] O. Carlton, *Cryptograms to Keep You Sharp*. New York: Sterling Publishing Company, Inc., 2002.

[2] E. Barker. “Recommendation for Key Management”, 2016, NIST Special Publication 800-57 Part 1 Revision 4.

[3] C. Pomerance. (1999, March 8). *A Tale of Two Sieves* [online]. Available: http://www.ams.org/notices/199612/pomerance.pdf

[4] C. Pomerance. (2008, April 30). *Smooth numbers and the quadratic sieve* [online]. Available: https://math.dartmouth.edu/ carlp/PDF/qs08.pdf

[5] (2017, May 4). *Express 4* [online]. Available: https://expressjs.com/

[6] (2017, May 4). *Node.js* [online]. Available: https://nodejs.org/en/

[7] (2017, May 4). *Heroku* [online]. Available: https://www.heroku.com/

[8] (2017, May 15). *Qt website* [online]. Available: https://www.qt.io/

[9] (2016, Dec 19). *The GNU Multiple Precision Arithmetic Library* [online]. Available: https://gmplib.org/

[10] "Crypto++® Library 5.6.5", https://cryptopp.com/

Glossary

Agile A software development methodology that focuses on iterative development and collaboration. 13

Crypto++ A C++ crypto library. Provides implementations of many encryption algorithms and hash functions. 8, 14

cryptogram A game where the player tries to decrypt a message. The message is usually encrypted with a substitution cipher. 4

GMP GNU Multiple Precision arithmetic library. Provides a big integer C++ class interface. 14

GUI Graphical User Interface. 14

Hash Function A hash function is a one way function whose output is indistinguishable from random. 4, 14

IDE Integrated Development Environment. A single program that handles many aspects of software development, which could include: compilation, GUI design, debugging tools, and other features. 14

Linux Mint A Ubuntu-based linux distribution. 15

NIST National Institute of Standards and Technology. 10

OpenCL Open Computing Language. A framework for performing computations on a variety of hardware, including field programmable gate arrays(FPGA) and specialized graphics cards (GPU). More info at https://www.khronos.org/opencl/. 13

Qt A cross platform graphics API. 9, 14

RSA A public key cryptosystem, invented in 1977. Widely used in internet security today. 4, 10

Scrum A type of Agile development. 14

Slack A messaging application built for team collaboration. 15

SSD System Sequence Diagram. 7, 13