Cracking RSA with Various Factoring Algorithms

Brian Holt

1 Abstract

For centuries, factoring products of large prime numbers has been recognized as a computationally difficult task by mathematicians. The modern encryption scheme RSA (short for Rivest, Shamir, and Adleman) uses products of large primes for secure communication protocols. In this paper, we analyze and compare four factorization algorithms which use elementary number theory to assess the safety of various RSA moduli.

2 Introduction

The origins of prime factorization can be traced back to around 300 B.C. when Euclid's theorem and the unique factorization theorem were proved in his work *Elements* [3]. For nearly two millenia, the simple trial division algorithm was used to factor numbers into primes. More efficient means were studied by mathemeticians during the seventeenth and eighteenth centuries. In the 1640s, Pierre de Fermat devised what is known as Fermat's factorization method. Over one century later, Leonhard Euler improved upon Fermat's factorization method for numbers that take specific forms. Around this time, Adrien-Marie Legendre and Carl Gauss also contributed crucial ideas used in modern factorization schemes [3].

Prime factorization's potential for secure communication was not recognized until the twentieth century. However, economist William Jevons antipicated the "one-way" or "difficult-to-reverse" property of products of large primes in 1874. This crucial concept is the basis of modern public key cryptography. Decrypting information without access to the private key must be computationally complex to prevent malicious attacks. In his book *The Principles of Science: A Treatise on Logic and Scientific Method*, Jevons challenged the reader to factor a ten digit semiprime (now called Jevon's Number)[4]. The number was 8,616,460,799. Modern computers can quickly find the factors of this number with even an inefficient algorithm like trial division. However, Jevons wanted to convey the contemporary difficulty of extracting two large primes without access to a machine.

In 1903, Jevon's Number was factored by mathematician Derrick Lehmer, who held an interest in number theory and prime factorization[5]. He used an algorithm later known as the continued fraction factorization method. The number was also resurfaced by mathmetician Solomon Golomb in 1996. Golomb proved that the number could be factorized in under a couple hours using only hand calculations with basic modular arithmetic known in the 1800s.[2]. He reduced the number of candidate squares from Fermat's factorization method by a factor of 20.

Between the time of Jevon and Golomb, prime factorization methods were further improved. In the 1970s, mathematician John Pollard developed factorization algorithms such as Pollard Rho and Pollard $P-1$ [6]. These only rely on elementary number theory (Fermat's Little Theorem). Fermat's Little Theorem states that $a^p \equiv a \mod p$ for all integers a and all primes p. There is a special case of the theorem which states that $a^{p-1} \equiv 1 \mod p$ if a if $a \mod p \neq 0$. In 1974, Pollard created his $P-1$ method. Originally, the algorithm used an input bound B, which describes the point at which the algorithm will stop attempting to factor the semiprime M. Next, we calculate an exponent K equal to the products of prime numbers raised

to powers less than the bound B. Let a be an integer coprime to M. Taking the greatest common denominator of $a^K - 1$ and the semiprime M may or may not yield the factor p. If the exponent K happens to be a large factor of $p - 1$ for a semiprime $M = pq$, then we can extract the prime factor P by Fermat's Little Theorem. The factorization time of this algorithm is proportional to the bound B and size of the semiprime M. Larger boundaries greatly increase the factorization time. His Rho algorithm was conceived one year later in 1975 and was the quickest factorization method at the time. It uses concepts from elementary probability (birthday paradox) and works with only two numbers in memory. The birthday paradox describes the surprisingly high 50% probability of two matching birthdays appearing in a room with 23 random people. This is because the probability is based on the large number of possible birthday pairs for 23 people. The runtime of Pollard Rho is proportional to the square root of input semiprime M. Depending on the implementation, failure may be returned by either method.

During this decade, the problem of extracting large prime factors was incorporated into a new encryption scheme. In 1977, computer scientists Ronald Rivest, Adi Shamir, and Leonard Adleman worked together to define a new cryptosystem based on prime factorization [1]. The public key is the semiprime M (called the modulus) and an e relatively prime to $\phi(M) = (p-1)(q-1)$. A private key $d \mid ed \equiv 1 \mod \phi(M)$, only known by the messenger and recipient, can be used to encrypt a private message m^e. The recipient may then unlock the message using $(m^e)^d$. This requires an attacker to factorize $N = pq$ to be able to unlock the secret message. The scheme, titled RSA, is still implemented in most electronic payment and communication platforms today. If factoring is easy, then RSA can be cracked. Whether the converse of this statement holds true or not is currently unknown. It is believed that even if RSA is crackable, factoring remains computationally difficult.

We study the performance of four factorization algorithms: Trial Division, Fermat, Pollard Rho, and Pollard $P - 1$. We also define and analyze a new algorithm based by gerenalizing Solomon Golomb's techniques in his 1996 paper. Comparing the factorization times for each algorithm based on various moduli characteristics allows us to identify types of unsafe primes for practical encryption.

3 Methodology

I created five C programs to output the runtime of the aforementioned algorithms. Trial division attempts to divide the semiprime input M by every prime between 2 and $\sqrt{M}$, inclusive. Eventually, one factor will be found. The amount of steps required to reach a solution grows exponentially as the length of M grows. Although this ancient method is straightforward, it is the second least efficient algorithm known, falling behind random guessing[3]. It was not until the 1700s when Fermat greatly reduced the time required to factor odd numbers. It is applicable to every semiprime where $pq = M$ where $p, q \neq 2$. He leveraged the fact that all odd numbers are representable by the difference of two squares. The method begins at $a = \sqrt{M}$ and increments a by one for every step. A factor $(a - b)$ is found when $b^2 = a^2 - M$ is a perfect square. However, as Golomb noticed, Fermat's method uses more steps than necessary.

To assess the magnitude of Golomb's improvements on Fermat's Method in the timing experiment, one must first anaylze his techniques for factoring Jevon's number. Next, this approach will then be broken into multiple steps and combined into a generalized algorithm. The algorithm must work for any RSA modulus M. Additionally, we define an input to the algorithm N which may vary the factorization efficiency results.

Golomb factorizes Jevon's Number J faster than Fermat's method alone by eliminating candidate a for the equation $a^2 - b^2 = J$. The original algorithm takes an input $N = 100$ and semiprime $J = 8616460799$ and first calculates $D = (-J \equiv +1 \mod N)$. Next, he finds all of the unique squares $\mod N$ that differ by D. This produced pairs $(n_1^2, n_2^2) \mod N = (00, 01)$ and $(n_3^2, n_4^2) \mod N = (24, 25)$. He could deduce, therefore, that $a \mod N$ must equal $n_i \mod N$. This reduced the number of steps until success from 56 to 8, making his method 7x more efficient than Fermat!

Although Golomb's techniques were only used to factor Jevon's Number, I have generalized this approach into an algorithm that will factorize any semiprime input. To reduce the number of steps in Fermat's method to produce a factor of the semiprime M, we first find the value $D \equiv -M \mod N$. We then square every integer from zero to the input N and gather all the unique remainders when the squares are divided by N. This yields a list of unique $(i^2 \mod N) \mid i \in [0, N]$. Next, we search for all pairs of $i^2 \mod N$ in the list that are separated the difference D, and add them to a separate list. Finally, we then find all $i \in [0, N]$ whose squares are included in this final list. This means that all i will then satisfy $a \equiv i \mod N$. In other words, we only have to try the number of i for every would-be N increments of a using Fermat's method! The formalized algorithm is shown below.

Algorithm 1: Golomb's Technique (integer N, integer semiprime)

```
Result: One factor of the semiprime input
all_square_endings = [ ];
target_square_endings = [ ];
a_square_endings = [ ];
a_endings = [ ];
b^2 = 0;
temp = 0;
a_base = √(semiprime);
difference = (semiprime mod N) −N;
for i = 0 through N do
    temp = i^2 mod N;
    if a not in all_square_endings then
        append temp to all_square_endings;
    end
end
for i = 0 through length (all_square_endings) do
    temp = ( all_square_endings[i] + difference);
    if temp in all_square_endings then
        append temp to a_square_endings;
        append all_square_endings[i] to a_square_endings;
    end
    temp = (all_square_endings[i] − difference) mod N;
    if temp in all_square_endings then
        append temp to a_square_endings;
        append all_square_endings[i] to a_square_endings;
    end
end
for i = 0 through N do
    temp = i^2 mod N;
    if temp in a_square_endings then
        append temp to a_endings;
    end
end
while a_base != lcm(a_base , N) do
    a_base =a_base - 1;
end
b^2 = a_base^2 - N;
while true do
    for i = 0 through length(a_endings) do
        b^2 = (a_base + a_endings[i])^2 − N;
        if b^2 is perfect square then
            return
        end
        a_base + a_endings[i] − √(b^2);
    end
    a_base += N;
end
```

Finally, Pollard Rho and Pollard $P - 1$ leverages other basic number theory concepts. These algorithms combine simple math functions such as modular exponentiation, random number selection, and greatest common denominator. My implementation of $P - 1$ uses a fixed $a = 2$ and calculates:

$$gcd(a^K - 1, M) \mid \forall K \leq \prod_{\text{primes } q \leq B = 17} p^{\lfloor log_q B \rfloor}$$

or until a factor is found. The algorithm should succeed if the factors of $p-1$ are less than $12,252,240 = 2^4 \times 3^2 \times 5 \times 7 \times 11 \times 13 \times 17$. Failure is returned otherwise. Next, my Pollard Rho implementation will run until a factor of M is produced. It begins with four variables: $x = 2, y = 2, d = 1$, and a counter $c = 1$. We set x equal to the

output of a pseudorandom function $g(x) = x^2 + c \mod M$. We then set $y = g(g(y))$ and d equal to the greatest common denominator between the distance $|x - y|$ and M. This will repeat until $d \neq 1$. Normally, if $d = n$, failure is returned. Instead, my implementation of Pollard Rho will increment the counter c until the factor is found. Other implementations of these well-known algorithms can also be found on the internet. Failure trials of $P - 1$ are not considered in the timing results.

To time each factorization attempt, I used the standard C libraries. Because many tested semiprimes exceeded the maximum size of `longlongint`,I used the LibToMMath library to implement each algorithm. This library supports a rich variety of mathematical functions for multiple-precision integers. During testing, most timing runs above 100 seconds were thrown away. Following the testing phase, the Python library MatPlotLib was used to represent the extracted data.

The first test did not depend on the factorization time. Instead, an additional program was written to derive the efficiency of 100 Golomb inputs $N \in [1000, 100000]$. I define the efficiency to equal N, the potential amount of Fermat increments, divided by the number of candidate a. For example, Golomb's original $N = 100$ algorithm only required 5 tries for every 100 Fermat increments, resulting in an efficiency of $\frac{100}{5} = 20$. For the algorithm to work properly for all N and semiprime input, extra candidate a were necessary to test. Hence, the actual efficiency tends to be much lower than what is possible. The program was run for all RSA moduli tested in the project. For each semiprime, the top 5 most efficient N were added to a list of frequencies. This allows us to understand which Golomb inputs could be used to factorize in the least amount of time.

Next, the results of each timing test were plotted against various characteristics of the modulus $M = pq$. This allows us to assess the performance of each algorithm and draw conclusions about safer pq. We first assessed which Golomb input had the most efficient response to the size of M. Next, we compared the semiprime size response for all of the algorithms, including Golomb $N = 1000$ and the efficient Golomb method. The sizes of the RSA moduli factored ranged between 10^9 and 10^{28}. RSA outside of this range were either solved instantaneously or could only be factored by Pollard $P - 1$ in less than the maximum time bound. Next, we analyzed the response of each algorithm to the smoothness of $p-1$. The smoothness of each semiprime is defined to be the maximum prime factor of the totient function $\phi = p-1$. Finally, we compared each algorithm to the relative distance $|p - q|$.

4 Results

First, I attempted to identify N for which Golomb's techniques work well. I also tried to find charactersitics of semiprimes that can be factored well for certain N. During testing, I noticed a strange phonomenon where seven or eight Golomb algorithm input N were consistently efficient at factoring. These inputs were exceptionally efficient for all primes. In other words, there were no trends between primes and efficiency for input N. Out of these, $N = 36000, 63000, 72000, 84000, 99000$ appeared the most. Honorable mentions include $N = 54000$ and $N = 78000$, which also tended to have drastic efficiency spikes. See figure 1 on the next page for the results of the Golomb efficiency test. These inputs were occasionally between 8 to 13 times as efficient as $N = 1000$ or even $N = 100000$!

I was surprised by the small differences when increasing the magnitude of the N input. Initially, I predicted that increasing N from 1000 to 10,000 or more would greatly improve the efficiency. I suspected this because analyzing more numbers should theoretically yield less candidate a. Instead, each input $N = 1000, 10000, 100000$ had relatively low but similar efficiencies. I believe that this is due to the added

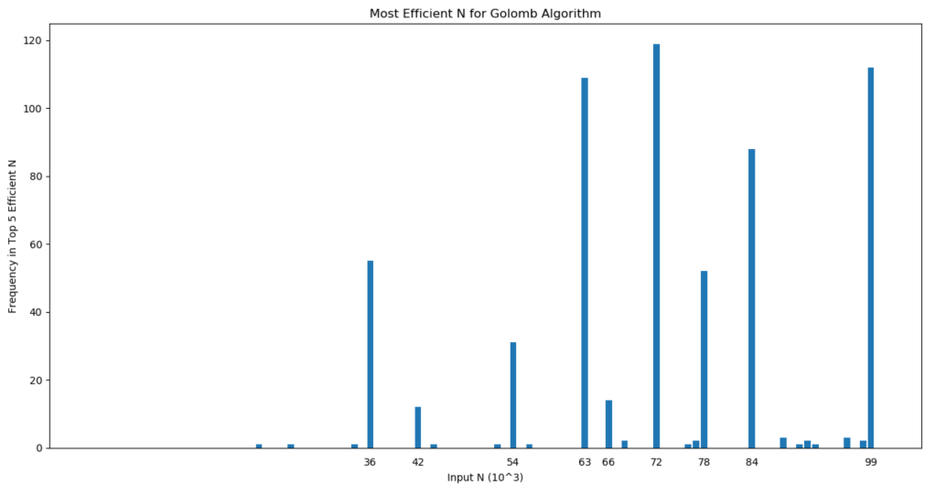

Figure 1: Frequencies of the top five most efficient N inputs for the Golomb algorithm for all tested RSA moduli.

"safety" of the generalized algorithm, which calculates more a than necessary to guarantee a factor output for any N or semiprime input. Further calculations or program enhancements may be necessary to reduce candidate a. A future goal is to streamline the algorithm to make it equally or more efficient than Golomb's original techniques. In his paper, he futher improves the efficiency by raising N to 1000, filtering even more candidate a [2]. This reduces the totoal number of steps until success from 56 to 8 to 3. Perhaps a super-efficient method may perform upfront calculations for a long list of very efficient N, thereby minimizing the amount of Fermat increments necessary to find a factor p. I also aim to find why specific inputs tend to yield such higher efficiencies than other N.

During the next test, I empirically assessed which of the N inputs were the most efficient, using Golomb 1000 as a baseline. For smaller M, the runtime of the algorithm is proportional to N. The timing results prove that Golomb 1000 produces the fastest runtime until the size of the modulus exceeds the 10^{18} range. See figure 2 for the response of the Golomb algorithm to various N and M size. At this point, the input with the lowest runtime quickly shifts to $N = 99000$. Occasionally, we observe that $N = 36000$ outperforms other N. I hypothesize that $N = 1000$ tends to have a lower runtime for smaller M due to the fact that there is less upfront calculation compared to $N = 99000$. However, the performance of $N = 99000$ for semiprimes greater than 10^{18} demonstrates that the upfront calculations are well worth the wait for larger p, q. Hence, for selected N, more calculations performed intially increases the efficiency and the overall runtime of Fermat. This supported trend I predicted before the experiment. For this reason, I included the results of both Golomb $N = 1000$ and $N = 99000$ for the other response comparisons conducted below.

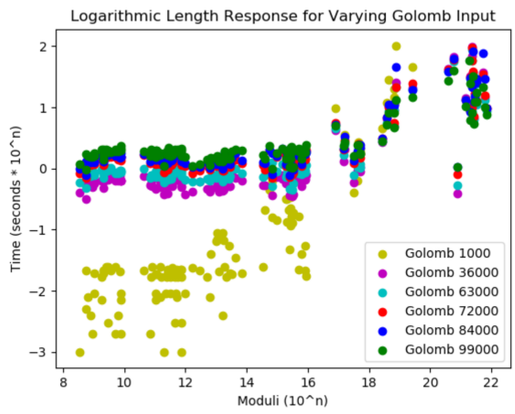

Figure 2: Logarithmic time response for the most efficient Golomb N input and $N = 1000$ to moduli size.

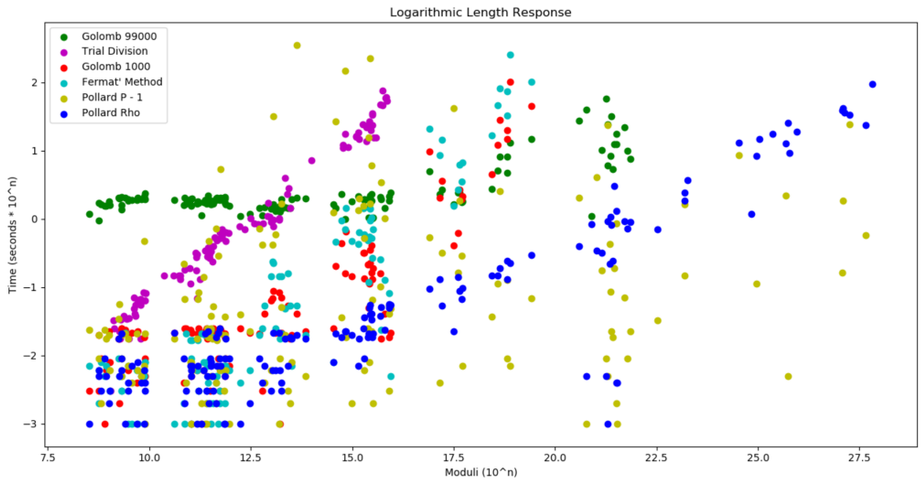

Figure 3: Logarithmic time response of each algorithm to semiprime size.

We compared the timing response of all algorithms to the size of the modulus M. It was evident that most algorithms required a longer time to find the factor when p and q were larger. Figure 3 depicts the logarithmic response of each method to increasingly large p and q. Trial division had a nearly perfect exponential time response to the size. Fermat's method shared an exponential shape with trial division, but was offset by an order of magnitude. The slopes of trial division, fermat, and the Golomb methods were similar, suggesting that each algorithm has about the same growth rate with respect to p and q. Golomb's $N = 1000$ input outperformed Fermat's method for $10^{13} \leq M \leq 10^{18}$, at which point $N = 99000$ became the most efficient Golomb algorithm. We also observe that all Fermat and Golomb 1000 factorization times exceeded 100 seconds for $M > 10^{19}$. This point was not reached by Golomb 99000 until $M > 10^{22}$. Hence, by carefully eliminating candidate a, Golomb's techniques can beat Fermat's method by a few orders of magnitude for pq! Seeing the result of Golomb's techniques for higher N may prove that upfront calculations could existing existing factorization algorithms based on Fermat's method. With the expection of a few outliers, Pollard Rho also had an exponential growth rate. However, the logarithmic slope was much lower than the other methods. This shows that Pollard Rho has the best response to the size of the semiprime out of the algorithms tested. Except for a slightly higher tendency to fail for greater M, Pollard $P-1$ runtimes appear to have no relationship to p or q size. To these ends, the size of the primes used in the RSA modulus is highly likely to increase the time required for a factoring attack.

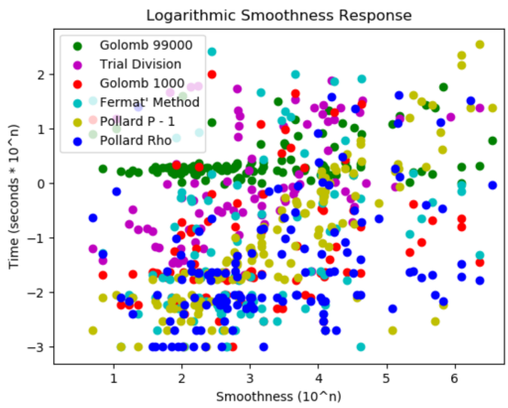

Figure 4: Logarithmic time response of each algorithm to the smoothness of $\phi(p) = p - 1$.

During the next test, we calculated the maximum prime factor of $p-1$ to compare the runtimes to the smoothness of $\phi(p)$. See figure 4 for the smoothness reponse of each algorithm. The performance of Pollard $P-1$ exemplified a direct exponential growth trend as $\phi(p)$ increased. This is because of the method's dependence on

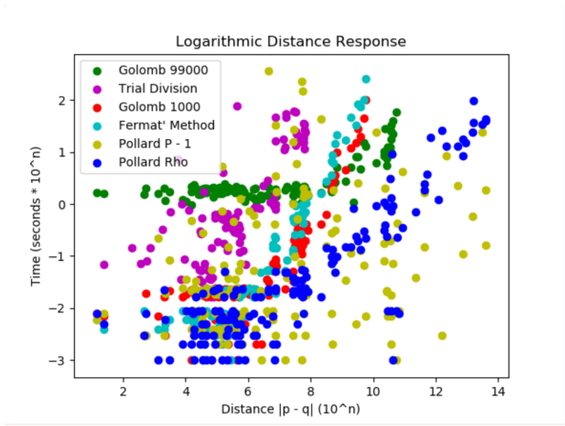

Figure 5: Logarithmic time response of each algorithm to the relative distance $|p - q|$.

Fermat's Little Theorem, which states that $a^{p-1} \equiv 1 \mod p$, where a is coprime to a prime p. Hence, if $i = p - 1$, then the greatest common denominator of $2^i - 1$ and the semiprime M will produce p. If M has a $p - 1$ or a $q - 1$ with a small maximum factor, then Pollard's $P-1$ algorithm will work very fast. However, as this smoothness value increases, the runtime grows exponentially due to the efficiency's dependence on the bound B. The smoothness of various $p - 1$ had no observable influence on any of the other algorithms tested.

We finally examined at the respone of the algorithms to various distances $|p-q|$. Figure 5 displays the logarithmic processor time of each algorithm against the distance. The trial division and Pollard $P-1$ algorithms seemed to have no relationship to the distance $p - q$. However, Fermat's Method, Golomb's techniques, and Pollard Rho all seemed to have an exponential growth for factorization runtime as the distance increased. This is most likely because Fermat's method begins at $a = \sqrt{M}$ and gets incremented. If $(a + b)$ and $(a - b)$ are close together, then a will only need to be incremented relatively few times until a factor is found. This translates to an exponentially smaller runtime. Several outliers were identifiable in the data for Pollard Rho, so the correlation between M size and the Pollard Rho runtime is stronger than that of the distance.

5 Conclusion

From the aforementioned results, we are able to better understand the strengths of each algorithm and thus determine features of safe $M = pq$ for practical RSA implementations. We see a direct correlation between factoring time and semiprime size for five of the six algorithms tested. Hence, a good encryption scheme should use p and q as large as possible. Additionally, the results suggest that some of the algorithms perform exceptionally well for smaller values of $|p - q|$. These include Fermat's method (with or without Golomb's techniques) and Pollard Rho. A safe moduli should have primes factors that are relatively far. Finally, the smoothness test indicates that there is a relationship between the largest prime factor of $\phi(p) = p - 1$ and the efficiency of the Pollard $P - 1$ factoring algorithm. To prevent suspectibility to this factoring attack, the largest prime factor of $p-1$ both $q-1$ must be very large in proportion to p and q. These traits are easier to satisfy if larger prime numbers are used. Ultimately, by avoiding unsafe p and q, the algorithms studied in this paper will be unable to quickly factor the semiprime public key. Carefully choosing secure RSA moduli greatly increases the time needed for an attacker to decrypt private messages.

In the future, I plan to continue this project by identifying additional characteristics of safe pq. I will research the time response of other more sophisticated algorithms to pq traits in this paper. I also aim to study trends of the Golomb algorithm. My hope is to streamline my current program to further reduce the number of candidate a. I will also continue the search for anomaly N inputs for values outside of the 10^4 and 10^5 range. Additionally, I hope to test intermediate values rather than those for which $N \equiv 0 \mod 1000$.

References

[1] Michael Calderbank. "The RSA Cryptosystem: History, Algorithm, Primes". In: (2007).

[2] Solomon Wolf Golomb. "On Factoring Jevons' Number". In: *Cryptologia* 20.3 (1996), pp. 243–246.

[3] Per Leslie Jensen. "Integer Factorization". University of Copenhagen, 2005. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.693.9784&rep=rep1&type=pdf.

[4] William Stanley Jevons. *The Principles of Science: A Treatise on Logic and Scientific Method*. 1873.

[5] Derrick Norman Lehmer. "A Theorem in the Theory of Numbers". In: *Bulletin of the American Mathematical Society* 13.10 (1907), pp. 501–502.

[6] Samuel S. Wagstaff. *The Joy of Factoring*. Vol. 68. Providence, Rhode Island: American Mathematical Society, 2013, pp. 135–140. DOI: http://dx.doi.org/10.1090/stml/068.