Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Implementation of Maurer's method for prime generation

Bachelor Thesis

Daniel Frey

September 18, 2020

Advisors: Prof. Dr. Kenneth Paterson, Mia Filić

Department of Computer Science, ETH Zürich

Abstract

Prime numbers are used in cryptographic protocols due to their unique number-theoretic properties.

Given an interval, a Prime Generation Algorithm (PGA) produces a random prime from the given interval. This bachelor's thesis focuses on the PGA presented originally by Ueli Maurer in 1995, which generates provable primes.

In this thesis, the mathematical background of the algorithm is presented and the underlying lemmas are proven in detail. We implement the algorithm with its key functions in the programming language C and adjust its parameters using experiments. Additionally, we compare our implementation with the prime generation function implemented in OpenSSL with respect to their running times.

If we lessen the diversity of reachable primes and modify our implementation accordingly, we can significantly decrease its expected running time. However, the unmodified version as well as the modified one is in expectation slower than the OpenSSL PGA implementation.

Contents

Contents **iii**

1 Introduction **1**

2 Number-Theoretic Preliminaries **3**
- 2.1 Notation . . . 3
- 2.2 Prerequisites . . . 3
- 2.3 Lemmas . . . 4

3 Algorithm **11**
- 3.1 Description of the Algorithm (RandomPrime) . . . 11
  - 3.1.1 Base Case . . . 11
  - 3.1.2 Recursion Case . . . 13
- 3.2 Function `GenerateSizeList` . . . 13
  - 3.2.1 Simplified Version . . . 15
- 3.3 Function `CheckLemma1` . . . 16
- 3.4 Function `PrimeTest` and Function `TrialDivision` . . . 16
- 3.5 Correctness of RandomPrime . . . 17

4 Implementation **19**
- 4.1 Libraries . . . 19
  - 4.1.1 OpenSSL . . . 19
  - 4.1.2 GNU Multiple Precision Arithmetic Library . . . 19
- 4.2 Implementation of `GenerateSizeList` . . . 20
- 4.3 Trial Division Bound . . . 21
- 4.4 Implementation of RandomPrime . . . 23

5 Running Time Experiments **25**
- 5.1 Prime Number Generator in OpenSSL . . . 25
- 5.2 Experimental Results . . . 27
- 5.3 Runtime Modification . . . 27

6 Discussion **31**
6.1 Advantages and Disadvantages of RandomPrime 31
6.2 Further Modifications 33

7 Conclusion **35**

A randomprime.c **37**
A.1 Installation of OpenSSL 3.0.0 on Euler 37
A.2 Compilation on Euler 37
A.3 Source Code 37

Bibliography **45**

Chapter 1

Introduction

Nowadays cryptographic tools are used nearly everywhere by everyone, even if most people are not aware of it. Cryptographic protocols are key components in communication applications, online banking, electronic voting and much more. In many of these cryptographic protocols prime numbers are crucial. Without primes and their properties, the protocols can be broken by adversaries who then can decrypt messages, steal money, manipulate elections and do other undesirable things.

RSA (Rivest–Shamir–Adleman) is an example of a widely used public-key cryptosystem in which primes are essential. RSA provides secure data transmission and utilises two prime numbers of same bit-length to generate the private key and the public key. The utilised primes are hundreds or even thousands of digits long and the product of the two primes is a publicly known value. If an adversary can factorise this product, the protocol can be broken. Since the computational power of (super-)computers is continuously increasing, brute force attacks are feasible on larger and larger keysizes, and therefore primes with a larger bit-length are preferable.

In the *Internet of things* there are many electronic devices that also participate in a protocol including primes, for example, because of their work with sensitive data, which requires secure data transmission. These devices need to be able to generate primes using their limited computational capacity.

In summary, the generation of primes is an important task in cryptography. It should be performed efficiently, using frequently limited computational resources and maintaining the uniform distribution over the set of primes that fulfil specific given properties, e.g. primes of a specific bit-length.

The goal of this thesis is to understand, implement and analyse the Prime Generation Algorithm (PGA) presented in [6] by Maurer. His method, as well as other PGAs, uses a *trial-and-error* approach.

A PGA that uses the *trial-and-error* approach works as follows. Firstly, a random odd number is chosen. Secondly, the number is tested for primality using a primality test. If the number is found prime, the algorithm outputs it and stops. Otherwise, other prime candidates are continually chosen and tested until a candidate passes the test. Since primes are not too seldom[1], with an efficient primality test, the *trial-and-error* PGA eventually finds a prime. Utilisation of probabilistic primality tests results in a negligible error that the outputted number is composite.

The Maurer's PGA starts by generating a random composite number n_0 by recursively generating its prime factors, and then selecting the odd prime candidate $n = n_0 + 1$. Candidate n is tested for primality using the knowledge of previously generated prime factors of n_0. If the number is found prime, the algorithm outputs it as proven prime and stops. Otherwise a new candidate is chosen by regenerating factors of n_0.

This thesis is structured as follows. Firstly, in Chapter 2, we present the number-theoretical background of Maurer's method and prove in detail the lemmas on which the algorithm is based. Secondly, in Chapter 3 we present in detail the description of the algorithm. After the description, in Chapter 4 we outline our implementation. Finally, in Chapter 5 we compare the running time of our implementation with the PGA implemented in OpenSSL, and in Chapter 6 we discuss the results and list the advantages and disadvantages of the Maurer's PGA.

[1]According to the prime number theorem, for large enough N, the probability that a random integer not greater than N is prime is close to $1/\log(N)$.

Chapter 2

Number-Theoretic Preliminaries

In this chapter we present and prove necessary number-theoretic preliminaries previously presented in [6, pp. 126–129]. They will occur at several points in the thesis, i.e. when considering the correctness and the runtime of the RandomPrime algorithm in Chapter 3.

2.1 Notation

In this thesis $\mathbb{Z}_n^*$ denotes the multiplicative group modulo n and $\operatorname{ord}_n(x)$ denotes the order of x in $\mathbb{Z}_n^*$, i.e. the smallest positive integer t satisfying $x^t \equiv 1 \pmod{n}$.

Here we introduce the notion of the colon symbol (:) usage for a description of a set S with respect to its elements. This notation prevents a confusion with the symbol '|' used for the divisibility. As an example,

$$S = \left\{ x \in \mathbb{Z}_p^* : d \mid \operatorname{ord}_p(x) \right\}$$

describes the set of all numbers x in $\mathbb{Z}_p^*$ such that a number d divides $\operatorname{ord}_p(x)$.

With $\varphi(n)$ we denote Euler's totient function, i.e. the number of positive integers smaller than n and relatively prime to n, with the exception $\varphi(1) = 1$.

2.2 Prerequisites

This section encompasses several number-theoretic facts that will be used in the proofs of the subsequent section.

We start with three basic facts about the order in the multiplicative groups. Let $x \in \mathbb{Z}_m^*$. Then

$$n \mid m \quad \Rightarrow \quad \operatorname{ord}_n(x) \mid \operatorname{ord}_m(x), \tag{2.1}$$

$$x^n \equiv 1 \pmod{m} \quad \Rightarrow \quad \mathrm{ord}_m(x) \mid n, \tag{2.2}$$

and

$$x^l \equiv x^m \pmod{n} \quad \Rightarrow \quad l \equiv m \pmod{\mathrm{ord}_n(x)}. \tag{2.3}$$

Proofs of (2.2) and (2.3) can be found in [1].

The following two implications are facts about relatively prime numbers. Let $a, b, c \in \mathbb{N}$ and let a be relatively prime to b.

$$a \mid bc \quad \Rightarrow \quad a \mid c, \tag{2.4}$$

and

$$a \mid c \wedge b \mid c \quad \Rightarrow \quad ab \mid c. \tag{2.5}$$

Expression (2.4) is a generalization of Euclid's first theorem [3, p. 25]. Next, we present several basic facts about Euler's totient function.

Firstly,

$$\frac{\varphi(n)}{n} = \prod_{p|n}\left(1 - \frac{1}{p}\right) \geq 1 - \sum_{p|n} \frac{1}{p} \tag{2.6}$$

where the product and summation are over all distinct prime numbers dividing n. Secondly,

$$\varphi(ab) \geq \varphi(a)\varphi(b) \tag{2.7}$$

with equality if and only if $\gcd(a, b) = 1$. Thirdly,

$$\sum_{d|n} \varphi(d) = n. \tag{2.8}$$

And lastly, if p is prime, then the group $\mathbb{Z}_p^*$ is cyclic and

$$\left|\left\{x \in \mathbb{Z}_p^* : \mathrm{ord}_p(x) = d\right\}\right| = \varphi(d) \tag{2.9}$$

for every divisor d of $p - 1$.

2.3 Lemmas

In Chapter 3 we will present an algorithm for generating a random prime number of specific bit-size. The correctness of the algorithm lies upon Lemma 1, which is based on the Pocklington[1] theorem which can be found in [10, p. 121]. Lemma 1 ensures that a number $n = 2RF + 1$, generated by the algorithm, is prime.

[1]H. C. Pocklington (1870–1952) was an English physicist and mathematician.

Lemma 1 Let $n = 2RF + 1$ where the prime factorization of F is $F = q_1^{\beta_1} q_2^{\beta_2} q_3^{\beta_3} \cdots q_r^{\beta_r}$. If there is an integer a satisfying

$$a^{n-1} \equiv 1 \pmod{n} \tag{2.10}$$

and

$$\gcd(a^{(n-1)/q_i} - 1, n) = 1 \tag{2.11}$$

for $i = 1, \ldots, r$, then each prime factor p of n is of the form $p = mF + 1$ for some integer $m \geq 1$. Moreover, if $F > \sqrt{n}$, or if F is odd and $F > R$, then n is prime.

Example For $n = 62791$, we find that $n = 2R(3 \cdot 5 \cdot 7 \cdot 13) + 1$. Hence the prime factors of F are

$$q_1 = 3, \quad q_2 = 5, \quad q_3 = 7, \quad q_4 = 13.$$

For these assignments of n and F, $a = 3$ fulfils the conditions of Lemma 1 and we have that $F = 2 \cdot 3 \cdot 5 \cdot 13 > \sqrt{n}$. Therefore, using Lemma 1 with $a = 3$, we prove that n is prime.

Proof Let p be an arbitrary prime dividing n and let a be an integer satisfying both conditions (2.10) and (2.11) of Lemma 1. From (2.1) it follows that $\operatorname{ord}_p(a) \mid \operatorname{ord}_n(a)$ and from (2.2) we have that $\operatorname{ord}_n(a) \mid n - 1$. Therefore,

$$\operatorname{ord}_p(a) \mid n - 1. \tag{2.12}$$

Since p divides n, we derive from condition (2.11) that for all $i \in \{1, 2, \ldots, r\}$

$$\gcd(a^{(n-1)/q_i} - 1, p) = 1,$$

and thus

$$a^{(n-1)/q_i} - 1 \not\equiv 0 \pmod{p}.$$

Therefore, for all $c \in \mathbb{N}$,

$$a^{(n-1)/q_i} \not\equiv (a^{\operatorname{ord}_p(a)})^c \pmod{p}.$$

Hence $(n-1)/q_i \neq c \cdot \operatorname{ord}_p(a)$, which can be also written as

$$\operatorname{ord}_p(a) \nmid (n-1)/q_i. \tag{2.13}$$

From (2.12) and (2.13) it follows that there exists some $x \in \mathbb{N}$ such that $x \cdot \operatorname{ord}_p(a) = n - 1$ and that for all $y \in \mathbb{N}$, $y \cdot q_i \cdot \operatorname{ord}_p(a) \neq n - 1$. Thus, we get the following.

$$\begin{aligned} &\exists x \forall y: \quad y \cdot q_i \cdot \mathrm{ord}_p(a) \neq x \cdot \mathrm{ord}_p(a) \quad \wedge \quad x \cdot \mathrm{ord}_p(a) = n-1 \\ \Leftrightarrow \quad &\exists x \forall y: \quad y \cdot q_i \neq x \quad \wedge \quad x \cdot \mathrm{ord}_p(a) = n-1 \\ \Leftrightarrow \quad &\exists x: \quad q_i \nmid x \quad \wedge \quad x \cdot \mathrm{ord}_p(a) = n-1, \end{aligned}$$

which implies $\gcd(q_i, x) = 1$ as q_i is prime. Since $q_i^{\beta_i}$ divides $n-1$ and $q_i^{\beta_i}$ is relatively prime to x, we can use (2.4) to conclude that for all $i \in \{1, 2, \ldots, r\}$

$$q_i^{\beta_i} \mid \mathrm{ord}_p(a). \tag{2.14}$$

Implication (2.5) further implies that $F \mid \mathrm{ord}_p(a)$.

Since p is prime, $\mathrm{ord}_p(a)$ divides $p-1$. Therefore $F \mid p-1$, which implies that each prime factor p of n is of the form $p = mF + 1$ for some $m \in \mathbb{N}$. This proves the first part of the lemma.

Let now $F > \sqrt{n}$. As shown before, each prime factor p of n is greater or equal to $F+1$. Hence each prime factor of n is larger than $\sqrt{n}$. Since at most one prime factor of a number can be larger than its square root, it follows that n has only one prime factor and $p = n$. This completes the proof of the second part of the lemma.

To prove the last part of the lemma, we let F to be odd and $F > R$. It follows that $F+1$ is even and larger than 2. Furthermore, $F+1$ is neither prime nor a prime factor of n. Therefore the smallest prime factor of n is at least $2F+1$. Now we prove that n is prime by contradiction. Let n be composite. Then

$$2RF + 1 = n \geq (2F+1)^2 = 2(2F+2)F + 1.$$

It follows that $R \geq 2F + 2$ which contradicts $F > R$. □

Proving the primality of n using Lemma 1 requires the knowledge of some prime factors of $n-1$, such that $F > \sqrt{n}$, or that F is odd and $F > R$. Lemma 2 defined below presents the third sufficient condition and can be used in speeding up the Maurer's PGA presented in Chapter 3.

Lemma 2 Let n, R, F, and a be as in Lemma 1 and let $x \geq 0$ and y be such that $2R = xF + y$ and $0 \leq y < F$. If $F \geq \sqrt[3]{n}$ and if $y^2 - 4x$ is neither zero nor a perfect square, then n is prime.

Example Let $n = 62791$, as in the previous example for Lemma 1. In Lemma 1, F has to be greater than the square root of n to show that n is prime, whereas in Lemma 2, $F \geq \sqrt[3]{n}$ is enough. Therefore it is sufficient to find the following prime factors of $n-1$,

$$q_1 = 5 \text{ and } q_2 = 13,$$

since $F = 5 \cdot 13 \geq \sqrt[3]{n}$. For $x = 14$ and $y = 56$ such that $2R = xF + y$, $y^2 - 4x$ is neither zero nor a perfect square. Moreover, for these assignments of n and F, $a = 2$ fulfils the conditions of Lemma 1. Therefore n is proven to be prime using Lemma 1 and Lemma 2.

Proof Let $F \geq \sqrt[3]{n}$. From Lemma 1 we know that the smallest prime factor of n is greater or equal to $F + 1$. Therefore n has at most two prime factors because otherwise $(F+1)^3 \leq n$ which contradicts $F \geq \sqrt[3]{n}$. We show that if $y^2 - 4x$ is neither zero nor a perfect square, then n is prime by contradiction. Let n be composite. Then,

$$\begin{aligned} n &= (m_1F+1)(m_2F+1) \\ &= m_1m_2F^2 + m_1F + m_2F + 1 \\ &= \underbrace{(m_1m_2F + m_1 + m_2)}_{=2R} F + 1. \end{aligned} \tag{2.15}$$

Without the loss of generality, let $m_1 \leq m_2$. It follows that $m_1m_2 < F$, because $n \leq F^3$. Moreover, $m_1 + m_2 < F$, which is proven as follows.

> Since $1 \leq m_1 \leq m_2 \leq F - 1$ one of the following holds:
>
> 1. $2 \leq m_1 \wedge 2 \leq m_2 \quad \Rightarrow \quad m_1 + m_2 \leq m_1m_2 < F$
> 2. $m_1 = 1 \wedge m_2 < F - 1 \quad \Rightarrow \quad m_1 + m_2 < F$
>
> The case $m_1 = 1 \wedge m_2 = F - 1$ is not possible. In this case n would have been equal to $F^3 + 1$ which contradicts $n \leq F^3$.

Now, from (2.15) we have that $2R = xF + y$ for $x = m_1m_2$ and $y = m_1 + m_2$. Substituting m_2 by $y - m_1$ in $x = m_1m_2$ gives

$$m_1^2 - ym_1 + x = 0. \tag{2.16}$$

Using the quadratic formula to find a solution $m_1 \in \mathbb{N}$ of (2.16), we get that $y^2 - 4x$ needs to be either equal to zero or a perfect square. This finishes the proof. □

The usage of Lemma 1 to prove the primality of an integer n requires an $a \in \mathbb{Z}_n^*$ that fulfils the conditions of Lemma 1. Following lemmas demonstrate that such an a can be found easily when n is indeed prime and all prime factors of F are sufficiently large.

Lemma 3 Let p be a prime and let d be a divisor of $p - 1$. Then

$$\left|\left\{x \in \mathbb{Z}_p^* : d \mid \operatorname{ord}_p(x)\right\}\right| \geq \frac{\varphi(d)}{d}(p-1)$$

with equality if and only if $\gcd(d, (p-1)/d) = 1$.

Proof From (2.9) and $\mathrm{ord}_p(x) \mid p-1$ it follows that

$$\left|\left\{x \in \mathbb{Z}_p^* : d \mid \mathrm{ord}_p(x)\right\}\right| = \sum_{d' \,:\, d|d'|(p-1)} \varphi(d'). \tag{2.17}$$

On the right hand side of equation (2.17), every d' is of the form $d' = k \cdot d$ for some $k \in \mathbb{N}$. So, the sum can be rewritten as

$$\sum_{k \,:\, k|((p-1)/d)} \varphi(kd)$$

and is bounded from below using (2.7) as follows

$$\sum_{k \,:\, k|((p-1)/d)} \varphi(kd) \geq \sum_{k \,:\, k|((p-1)/d)} \varphi(k)\varphi(d). \tag{2.18}$$

The equality in (2.18) holds if and only if

$$\gcd(k,d) = 1 \quad \text{for all } k \text{ such that } k \mid (p-1)/d,$$

which is equivalent to $\gcd(d, (p-1)/d) = 1$.

Finally, using (2.8) in (2.18) we obtain

$$\sum_{k \,:\, k|((p-1)/d)} \varphi(kd) \geq \sum_{k \,:\, k|((p-1)/d)} \varphi(k)\varphi(d) = \varphi(d)\frac{p-1}{d}$$

which finishes the proof. □

Lemma 4 Let $p = 2RF + 1$ be a prime such that $F = \prod_{i=1}^{r} q_i^{\beta_i}$, $F > R$, and $\gcd(2R, F) = 1$, where $q_1, \ldots, q_r$ are distinct primes. Then the probability that a randomly selected base $a \in \mathbb{Z}_p^*$ is successful in proving the primality of p by Lemma 1 is equal to $\varphi(F)/F$ which is at least $1 - \sum_{i=1}^{r} 1/q_i$.

Example For $n = p = 62791$ and $F = 3 \cdot 5 \cdot 7 \cdot 13$ as before, we get

$$\left|\left\{a \in \mathbb{Z}_p^* : a \text{ proves the primality of } p \text{ by Lemma 1}\right\}\right| = 26496.$$

Hence, for a uniformly at random chosen $a \in \mathbb{Z}_p^*$,

$$\Pr[a \text{ proves the primality of } p \text{ by Lemma 1}] \approx 0.42198$$

which is indeed equal to $\varphi(F)/F$ and greater than $1 - \frac{1}{3} - \frac{1}{5} - \frac{1}{7} - \frac{1}{13} \approx 0.247$. This means that we only have to test 2 or 3 candidates for a on average to prove the primality of $p = 62791$.

Proof Since p is prime, $a^{p-1} \equiv 1 \pmod{p}$ holds for every $a \in \mathbb{Z}_p^*$. This implies that every $a \in \mathbb{Z}_p^*$ satisfies condition (2.10) of Lemma 1.

From $\gcd(2R, F) = 1$ it follows that F is odd and that

$$F \mid \mathrm{ord}_p(a) \quad \Leftrightarrow \quad \forall i \in \{1, 2, \ldots, r\} : a^{(p-1)/q_i} \not\equiv 1 \pmod{p}, \tag{2.19}$$

which we prove later on. Since p is a prime, the right-hand side of (2.19) is equivalent to condition (2.11) on a of Lemma 1. Therefore, for a uniformly at random chosen $a \in \mathbb{Z}_p^*$,

$$\Pr[a \text{ proves the primality of } p \text{ in Lemma 1}] = \Pr[F \mid \mathrm{ord}_p(a)].$$

By setting $d = F$ in Lemma 3 the upper probability is equal to $\varphi(F)/F$. From (2.6) it follows that

$$\Pr[a \text{ proves the primality of } p] \geq 1 - \sum_{i=1}^{r} \frac{1}{q_i}.$$

We have left out the proof of (2.19) which we now present. The left-handed implication ($\Leftarrow$) we have already proved in the proof of Lemma 1: From (2.10) and (2.11) we have derived (2.14) and hence also $F \mid \mathrm{ord}_p(a)$.

Next we prove the right-handed implication ($\Rightarrow$) by contradiction. Let $\gcd(2R, F) = 1$,

$$F \mid \mathrm{ord}_p(a) \tag{2.20}$$

and

$$a^{(p-1)/q} \equiv 1 \pmod{p} \tag{2.21}$$

for an arbitrary $q \in \{q_1, q_2, \ldots, q_r\}$. From (2.21) and (2.3) it follows that

$$(p-1)/q \equiv p - 1 \pmod{\mathrm{ord}_p(a)}.$$

Since $\mathrm{ord}_p(a) \mid p - 1$, we obtain

$$(2RF)/q \equiv 0 \pmod{\mathrm{ord}_p(a)}$$

and

$$\exists x : \quad (2RF)/q = x \cdot \mathrm{ord}_p(a). \tag{2.22}$$

From (2.20) and (2.22) we get

$$\begin{aligned} \exists x \exists y : &\quad (2RF)/q = x \cdot y \cdot F \\ \Rightarrow \exists x \exists y : &\quad 2R = x \cdot y \cdot q. \end{aligned}$$

Therefore $2R$ is a multiple of q and $\gcd(2R, F)$ can not be 1, which contradicts our assumption and thus proves the right-handed implication ($\Rightarrow$) of (2.19) when $\gcd(2R, F) = 1$. $\square$

Chapter 3

Algorithm

In this chapter we present the Maurer's PGA presented originally in [6, pp. 130–133] which is called RandomPrime algorithm. The first section outlines the algorithm in general form, while the subsequent sections detail its key functions. Lastly, we explain why RandomPrime satisfies the correctness property.

3.1 Description of RandomPrime

RandomPrime is a prime generation algorithm which takes as an input two numbers: P_1 and P_2 such that $P_1 < P_2$. If the interval $[\,P_1, P_2\,]$ is such that it contains a prime number, the algorithm outputs a prime $n \in [\,P_1, P_2\,]$. The pseudocode of RandomPrime is given with the Algorithm 1.

When using prime numbers in cryptographic protocols, we usually need a random prime n of a specific bit-size k. Therefore $[\,P_1, P_2\,]$ is often of the form $[\,2^{k-1}, 2^k - 1\,]$.

3.1.1 Base Case

RandomPrime is a recursive algorithm in P_2. If P_2 is smaller than a given constant P_0, the algorithm is in the base case of the recursion. In this case, the algorithm starts by choosing uniformly at random an integer $n \in [\,P_1, P_2\,]$ and then tests n for primality. If n is composite, another integer $n \in [\,P_1, P_2\,]$ is chosen uniformly at random and tested. The algorithm repeats this process until a prime n is found.

The primality test used above is captured with a function called `PrimeTest`. Since it will be used only on inputs n such that $n \leq P_2 < P_0$, the function needs to be efficient only on inputs smaller than P_0. A possible `PrimeTest` outline is given in Section 3.4.

Algorithm 1: RandomPrime(P_1, P_2)

```
Input  : two integers $P_1$ and $P_2$
Output: a random chosen prime number $n$ so that $P_1 \leq n \leq P_2$
Initialization of $P_0$ and $c_{int}$
if $P_2 \leq P_0$ then
    repeat
        $n =$ RandomInt( $P_1$ , $P_2$ )
    until PrimeTest( $n$ )
    return $n$
else
    $P =$ SquareRoot( $(P_1 - 1) \cdot (P_2 - 1)$ )  / 2
    $F = 1$
    $sizeList = factorList = [\,]$
    $r =$ GenerateSizeList($sizeList$)
    for $i = 0$ to $r$ do
        $Q =$ Exponentiate( $P$, $sizeList[i]$ )
        $factorList[i] =$ RandomPrime( $Q/c_{int}$ , $Q \cdot c_{int}$ )
        $F = F \cdot factorList[i]$
    end
    $I_1 = (P_1 - 1)/(2 \cdot F)$
    $I_2 = (P_2 - 1)/(2 \cdot F)$
    $success = false$
    repeat
        $R =$ RandomInt( $I_1$ , $I_2$ )
        $n = 2 \cdot R \cdot F + 1$
        if TrialDivision( $n$ ) then
            $a =$ RandomInt( 2, $n - 1$ )
            $success =$ CheckLemma1( $n$ , $a$ , $factorList$ )
        end
    until $success$
    return $n$
end
```

3.1.2 Recursion Case

If $P_2 \geq P_0$, the algorithm is in the recursion case. In this case, a prime $n = 2RF + 1$ is generated by recursively generating prime factors of F as follows.

Firstly, the geometric midpoint of the interval $[\,(P_1 - 1)/2,\ (P_2 - 1)/2\,]$, into which RF will fall, is computed:

$$P = \frac{\sqrt{(P_1 - 1) \cdot (P_2 - 1)}}{2} .$$

Secondly, using `GenerateSizeList` function, the number of prime factors of F and its relative sizes $s_1, s_2, \ldots, s_r$ are obtained. Thirdly, for each prime factor of F and its relative size s_i, the actual approximate size $Q_i = P^{s_i}$ is computed. Fourthly, for each $i \in \{1, 2, \ldots, r\}$, the algorithm selects a random prime factor q_i by calling recursively RandomPrime on $[\,Q_i/c_{int},\ Q_i \cdot c_{int}\,]$ where $c_{int} > 1$ is a small constant parameter. Produced prime factors $q_1, q_2, \ldots, q_r$ are stored in $factorList$.

After generating all prime factors q_i, $F = \prod_{i=1}^{r} q_i$ is fully defined. Next, the algorithm chooses R uniformly at random from the interval $[\,I_1, I_2\,]$ where $I_1 = (P_1 - 1)/\,2F$ and $I_2 = (P_2 - 1)/\,2F$, and computes a prime candidate $n = 2RF + 1 \in [\,P_1, P_2\,]$. This candidate n is first tested for primality by trial division and then by checking if the conditions (2.10) and (2.11) of Lemma 1 hold for a number a chosen uniformly at random from $[\,2, n-1\,]$. If n passes both tests, the algorithm stops and returns n. Otherwise, the algorithm repeats this procedure of selecting and testing R, n, a until $n = 2RF + 1$ is proven to be prime using Lemma 1.

3.2 Function `GenerateSizeList`

`GenerateSizeList` takes an empty list as an input and fills this list with the relative sizes $s_1, s_2, \ldots, s_r$ of the r largest prime factors $q_1, q_2, \ldots, q_r$ of a random number $x \in \mathbb{N}$, so that the unfactorized part of x does not contain a prime factor greater than q_r. We define the relative size of q_i with respect to x as

$$s_i = \frac{\log(q_i)}{\log(x)} ,$$

which is independent of the base of the logarithm. Additionally, the function has an output value, the number of generated relative sizes r. The relative sizes $s_1, s_2, \ldots, s_r$ are generated according to the following process.

First, `GenerateSizeList` chooses real numbers $u_1, u_2, \ldots, u_k, \ldots$ uniformly at random from

$$[0,1]\,,\ [0,1-u_1]\,,\ \ldots\,,\ \left[0,\, 1-\sum_{i=1}^{k-1} u_i\right]\,,\ \ldots$$

respectively. It stores numbers $u_1, u_2, \ldots$ in a list in decreasing order. Let s_k denote the k^{th} element of the list. The function stops generating $u_1, u_2, \ldots$ as soon as the first r numbers in the list are fixed. After choosing u_k, the values $s_1, s_2, \ldots, s_r$ are fixed if

$$s_r \ > \ 1 - \sum_{i=1}^{k} u_i. \tag{3.1}$$

Condition (3.1) (also mentioned in [6, p. 147]) ensures that $u_{k+1}, u_{k+2}, \ldots$ will be inserted into the list after s_r.

`GenerateSizeList` is called within the RANDOMPRIME algorithm to generate relative sizes of the prime factors $q_1, q_2, \ldots, q_r$ of F such that $n = 2RF + 1$. Thus, F is of a relative size $\sum_{i=1}^{r} s_i$ and R is of a relative size $1 - \sum_{i=1}^{r} s_i$. Therefore, `GenerateSizeList` needs to apply the stronger stopping criteria

$$s_r \ > \ 1 - \sum_{i=1}^{r} s_i \tag{3.2}$$

[6, p. 147] to fulfil condition $F > \sqrt{n}$ or $F > R$ of Lemma 1 and Lemma 4. Condition (3.2) ensures that R does not contain a prime factor greater or equal than q_r, which implies that distributions of relative sizes of prime factors of $(n-1)/2$ do not differ from those of a random integer. Moreover, it ensures that $\sum_{i=1}^{r} s_i$ is greater than $1 - \sum_{i=1}^{r} s_i$.

Thus, we define `GenerateSizeList` to generate $u_1, u_2, \ldots$ until (3.2) is fulfilled. Generated relative sizes $s_1, s_2, \ldots, s_r$ are then distributed according to $F_1, F_2, \ldots, F_r$, where

$$F_k(x) = 1 - \int_x^1 \left(F_k\left(\frac{t}{1-t}\right) - F_{k-1}\left(\frac{t}{1-t}\right)\right) \frac{dt}{t} \tag{3.3}$$

for $k \geq 1$ with $F_0(x) = 0$ for $x \in [0,1]$, and $F_k(x) = 1$ for $x \geq 1$ and $k \geq 1$ [6, p. 148]. Knuth[1] and Trabb Pardo[2] in [4] showed that F_k describes the distribution of the relative size of the k^{th} largest prime factor of a random number, i.e. $\Pr[s_k \leq x] = F_k(x)$, which proves the correctness of the outlined `GenerateSizeList` function.

[1]Donald E. Knuth (born in 1938) is an American computer scientist and mathematician.
[2]Luis Trabb Pardo is an Argentinian computer scientist.

Algorithm 2: `GenerateSizeList()`

```
sizeList = [ ]
c = 1
while True do
    u = uniformDistribution( 0, c )
    Append( sizeList, u )
    c = c − u
    ReverseSort( sizeList )
    for i = 0 to Length( sizeList ) do
        if sizeList[ i ] > 1− Sum( sizeList[ 0 : i ] ) then
            return sizeList[ 0 : i ]
        end
    end
end
```

3.2.1 Simplified Version

In [6, pp. 134–136], Maurer also explained a simpler version of the RandomPrime algorithm called FastPrime. In the simpler version, F is always generated using only one prime factor. Therefore, GenerateSizeList is modified to return only the relative size s_1 of the largest prime factor. The relative size s_1 is distributed according to F_1 in (3.3), which can be simplified for $1/2 \leq x \leq 1$ as follows[3]:

$$F_1(x) = 1 - \int_x^1 \frac{dt}{t} = 1 + \ln(x). \tag{3.4}$$

Equation (3.4) expresses the probability that the largest prime factor of a random number is of a relative size smaller than x, given that $1/2 \leq x \leq 1$. A value for s_1 can be generated using inverse transform sampling as

$$s_1 = F_1^{-1}(u) = e^{u-1}$$

where u is chosen uniformly at random from $[0,1]$, F_1^{-1} is the inverse function of F_1 and e denote the Euler number. The relative size s_1 is resampled if $F_1^{-1}(u)$ is smaller than $1/2$.

FastPrime has at most one recursive function call and therefore is faster than the more complex RandomPrime algorithm. On the other hand, it reduces the diversity of the generated primes by roughly 90 % [6, p. 136].

[3]We are only interested in $x \geq 1/2$ because we need $s_1 > 1 - s_1$ so that $F > R$.

3.3 Function `CheckLemma1`

`CheckLemma1` takes two integers n and a, and a not-empty list of integers $q_1, q_2, \dots, q_r$ as an input. The list elements $q_1, q_2, \dots, q_r$ are the prime factors of F such that $n = 2RF + 1$.

On a given input, the function starts by checking if condition (2.10) of Lemma 1 holds. According to Fermat's little theorem, if n is prime, every integer a fulfils (2.10). On the other hand, if n is composite, almost all candidates for a does not fulfil it. Next, `CheckLemma1` checks if the second condition (2.11) of Lemma 1 holds for every $i \in \{1, 2, \dots, r\}$. Assuming $F > \sqrt{n}$ or $F > R$ (and F is odd), if n is composite, every a that fulfils (2.10) does not fulfil (2.11) for at least one $i \in \{1, 2, \dots, r\}$.

When n and a pass all the checks, the function outputs *True*, indicating that input n is proven prime according to Lemma 1. As soon as one check fails, the function stops and outputs *False*, indicating that Lemma 1 can not be applied (on n with a) to derive primality of n. Hence, the output *False* does not say much about n. According to Lemma 4, for a prime n, the probability that a uniformly at random selected $a \in \mathbb{Z}_n^*$ is successful in proving the primality of n is equal to $\varphi(F)/F$, if $F > R$ and $\gcd(2R, F) = 1$.

Algorithm 3: `CheckLemma1`(n, a, $factorList$)

```
1 if $a^{n-1} \not\equiv 1 \pmod{n}$ then
2     return False
3 end
4 foreach $q \in factorList$ do
5     if Gcd( $a^{(n-1)/q} - 1$, $n$ ) $\neq 1$ then
6         return False
7     end
8 end
9 return True
```

3.4 Function `PrimeTest` and Function `TrialDivision`

`PrimeTest` and `TrialDivision` take an integer n as an input and check if n is prime using trial division.

By the construction of the algorithm RandomPrime, an input to `PrimeTest` is relatively small. Therefore, to test if n is prime, it is still efficient to just check if all integers $2 \leq x \leq \sqrt{n}$ do not divide n. Since a composite number must have a prime factor smaller or equal than its square root, it is not necessary to check if an integer $x > \sqrt{n}$ divides n. As soon as `PrimeTest` finds a factor $x \mid n$, the function stops and outputs n as proven composite. Otherwise, if the function does not find a factor of n, it outputs n as proven prime.

On the other hand, an input to the `TrialDivision` function can be arbitrary large. RandomPrime utilises `TrialDivision` to detect if a prime candidate n is divisible by any prime number greater than 2 and smaller than (or equal to) the trial division bound g. This avoids unnecessary function calls of `CheckLemma1`. `TrialDivision` is computationally much cheaper than `CheckLemma1`, and although it does not detect all composite numbers, it decreases the number of calls of `CheckLemma1` enough to effect and significantly reduce the running time of RandomPrime.

We select the trial division bound g so that for a generated prime candidate $n = 2RF + 1 \in [P_1, P_2]$, the expected runtime of `TrialDivision` followed by `CheckLemma1` is minimized. The value of g depends on the bit-size of n, the hardware, and the implementation of several basic arithmetic operations like multiplication and modular exponentiation. For the implementation of RandomPrime in Chapter 4, we determine g experimentally for different inputs $(P_1, P_2) = (2^{k-1}, 2^k - 1)$ where k is the bit-size of the output.

Algorithm 4: `TrialDivision`(n)

```
1 $i = 0$
2 repeat
3     if $n$ mod $primes[i] = 0$ then        ▷ $primes = \{3, 5, 7, 11, 13 \dots\}$
4         return False
5     end
6     $i = i + 1$
7 until $primes[i] > g$
8 return True
```

3.5 Correctness of RandomPrime

Since RandomPrime is a recursive algorithm, to prove the correctness of the algorithm it is sufficient to show that the base case is correct and the loop invariant holds. On a given input P_1 and P_2, the algorithm should output a random prime $n \in [P_1, P_2]$.

In the base case, the algorithm chooses candidates n from $[P_1, P_2]$ uniformly at random until `PrimeTest` declares a number n provable prime. Hence, it eventually returns a prime number chosen uniformly at random from the set of all primes in $[P_1, P_2]$, as required.

In the recursion case, prime candidates $n = 2RF + 1$ are generated bottom up by choosing a random even number $n_0 = 2RF$ from $[\,P_1 - 1,\, P_2 - 1\,]$. This number is generated in two steps:

1. Generate random prime factors of F of certain relative sizes by recursively calling RandomPrime.
2. Choose R randomly from a range such that $2RF + 1$ fall into $[\,P_1, P_2\,]$.

The relative sizes of prime factors of F are generated by `GenerateSizeList` which ensures that the resulting candidate $n = n_0 + 1$ is a random odd number from $[\,P_1, P_2\,]$.

After a prime candidate $n = 2RF + 1$ is generated, it is tested for primality using firstly `TrialDivision` and secondly `CheckLemma1`. When using `CheckLemma1` to test the primality of a number n, at least one of the following needs to hold:

$$F > \sqrt{n} \tag{3.5}$$

or

$$F \text{ is odd and } F > R. \tag{3.6}$$

In the algorithm, this is ensured by defining `GenerateSizeList` properly, i.e. as explained in Section 3.2. Condition (3.2) in `GenerateSizeList` ensures that R is smaller than the smallest prime factor of F. Thus, if F is odd, (3.6) is fulfilled. If F is even, the smallest prime factor of F is 2, and therefore R is approximately equal to 2. It follows that $F \gg 2R$, and (3.5) is fulfilled.

Since primality tests used are deterministic, a composite n is always detected. The algorithm generates fresh prime candidates by regenerating R until a prime is found.

Chapter 4

Implementation

In this chapter we present our implementation of the RandomPrime algorithm previously described in Algorithm 1. We meanly focus on the key functions of the algorithm and present statistical results about their output.

4.1 Libraries

Our implementation is written in the programming language C and utilises several basic functions of the software libraries OpenSSL (version 3.0.0 Alpha 4[1]) and GMP (version 6.1.2[2]).

4.1.1 OpenSSL

OpenSSL is a toolkit for several cryptographic applications. Its main purpose is to provide an open-source implementation of the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. Moreover, it is also a general purpose cryptographic library. OpenSSL is written in C and free available at [8].

Our implementation of RandomPrime uses the OpenSSL big-number data type, which allows us to perform arithmetic operations on integers of arbitrary size. Additionally, we use the random generator of the library to generate cryptographically strong pseudo-random numbers.

4.1.2 GNU Multiple Precision Arithmetic Library

GNU Multiple Precision Arithmetic Library (GMP) is an open-source library and part of the GNU project. GMP provides arithmetic operations on integers, rational numbers and floating-point numbers with arbitrary precision.

[1]released on June 25, 2020

[2]released in December, 2016

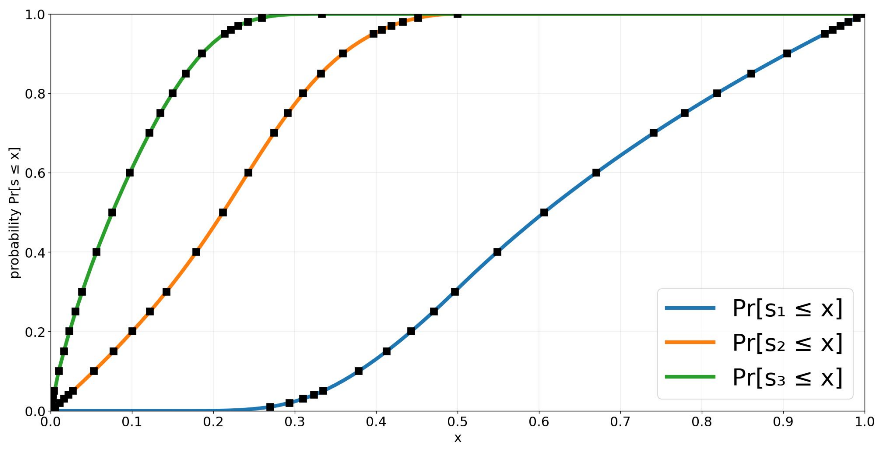


Figure 4.1: Distribution function estimates for $\Pr[s_1 \leq x]$, $\Pr[s_2 \leq x]$ and $\Pr[s_3 \leq x]$ derived from the output of 2^{22} calls of `GenerateSizeList` (solid lines), and values computed by Knuth and Trabb Pardo for F_1, F_2 and F_3 (black squares).

It is written in C, and suitable for use within several other programming languages utilising appropriate wrappers. Besides OpenSSL's integer type, in our implementation we also use GMP's integer type.

GNU Multiple Precision Floating-Point Reliable Library (GNU MPFR) is based on GMP and implements broader range of operations on floating-point numbers than GMP library. Our implementation of RandomPrime uses the GNU MPFR floating point number data type and its exponentiation function to calculate P^{s_i} in line 13 of Algorithm 1 (RandomPrime).

4.2 Implementation of `GenerateSizeList`

Function `GenerateSizeList` is implemented as explained in Section 3.2. It determines the number of prime factors of F such that $2RF + 1$ and their relative sizes $s_1, s_2, \ldots, s_r$ to RF. Figure 4.1 comprises the distribution function estimates for $\Pr[s_k \leq x]$ for $k \in \{1, 2, 3\}$ derived from the output of our implementation of `GenerateSizeList` and the values for F_1, F_2 and F_3 derived by Knuth and Trabb Pardo [4, p. 341]. They coincide with one another which suggests $\Pr[s_k \leq x] = F_k(x)$ and the correctness of our implementation.

The length of the generated list $s_1, s_2, \ldots, s_r$ sets the number of prime factors of F and the number of recursive calls of RandomPrime. In most cases F consists of only a few prime factors. After running the function 2^{30} times, we obtain an outputted list of size smaller than 3 in 82.70 % of the runs (Table 4.1). Therefore, only in 17.30 % of the cases, more than 2 recursive calls of RandomPrime were needed to obtain a value for F.

length r	1	2	3	4	5	6	7	> 7
probability in %	69.31	13.39	6.64	3.87	2.39	1.49	0.84	2.07

Table 4.1: The length r of the list generated by `GenerateSizeList` in probabilities derived from the output of 2^{30} function calls.

bit-size of n	trial division bound g
256	1913 (292th prime)
512	4481 (607th prime)
1024	12743 (1521th prime)
2048	37607 (3984th prime)

Table 4.2: Optimal trial division bounds in respect to the bit-size k of n. ($n \in [2^{k-1}, 2^k - 1]$)

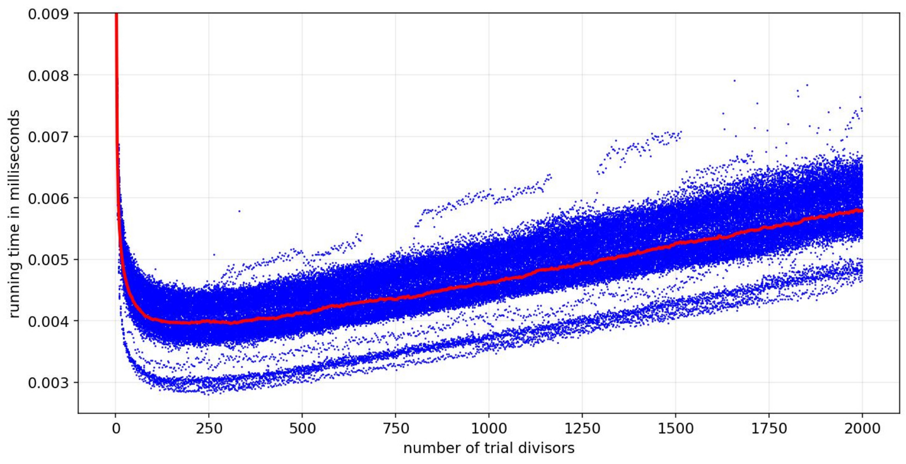
4.3 Trial Division Bound

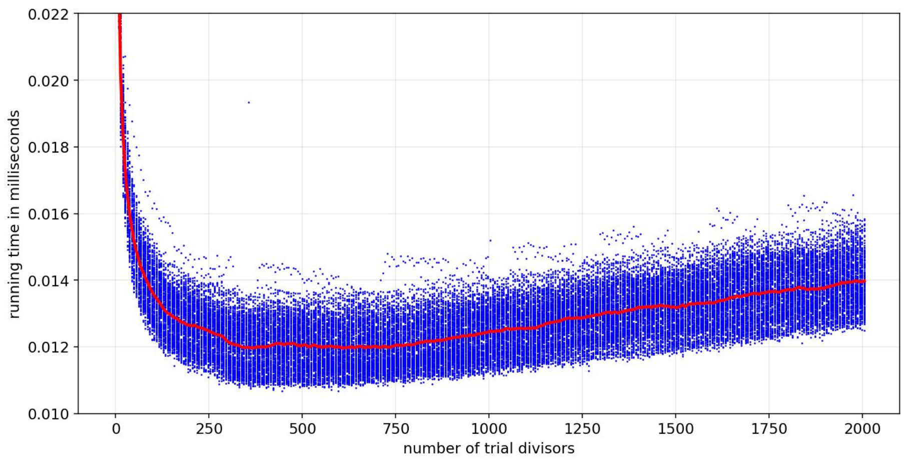
Function `TrialDivision` requires the determination of the trial division bound g. On the one hand, the trial division bound should be sufficiently large to detect as many composite candidates n in RandomPrime as possible (without utilising `CheckLemma1`). On the other hand, it should be small enough so that `TrialDivision` itself does not consume too much computational resources. For a fixed interval $[P_1, P_2]$ we follow the recommendations in [7, p. 146] and derive g experimentally by utilising Algorithm 5.

Algorithm 5 works as follows. Firstly, it generates a prime candidate n in the same way as RandomPrime in Algorithm 1. Secondly, n is tested with `TrialDivision`, and if needed, also with `CheckLemma1` for a uniformly at random selected integer $a \in [2, n-1]$. The running time of this test phase is measured and returned at the end.

We run Algorithm 5 for different g. In the first run of Algorithm 5, g is set to a small initial value g_0. We set $g_0 = 3$ and thus `TrialDivision` only tests if n is divisible by 3. In the next run, g is increased by a small number so that some more trial divisors are used within `TrialDivision`. We continuously increase g so that every time three, six or seven additional trial divisors are used (Figure 4.2). For each g, we run Algorithm 5 2^{20} times and derive the approximation of the expected running time of the primality testing phase. We continue the procedure until a strictly increasing tendency in the expected running time approximation is observed.

We set the trial division bound to the value for which the approximation of the expected running time is minimum. Table 4.2 lists the obtained optimal trial division bounds for $[2^{k-1}, 2^k - 1]$ for $k \in \{256, 512, 1024, 2048\}$. Therefore, we implement `TrialDivision` so that g is chosen depending on the input bit-size according to Table 4.2.





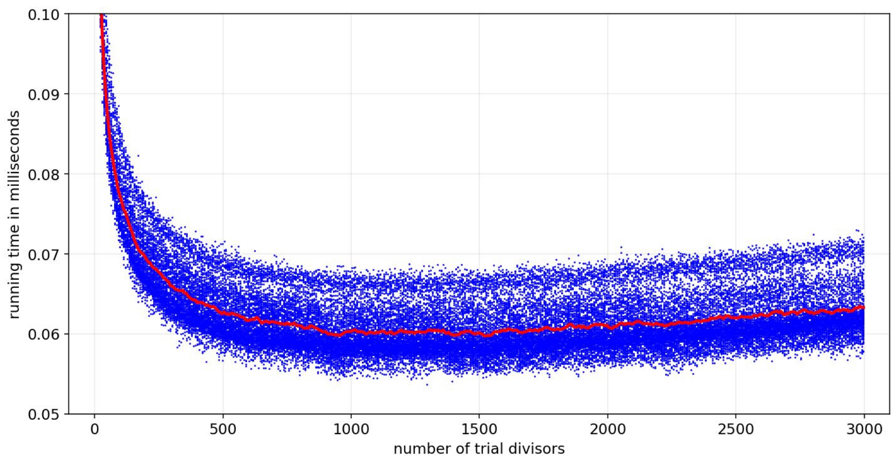


Figure 4.2: Expected running time of `TrialDivision` and `CheckLemma1` (in ms) in respect to the trial division bound for $[P_1, P_2] = [2^{255}, 2^{256} - 1]$ (the uppermost graph, step size = 3), $[P_1, P_2] = [2^{511}, 2^{512} - 1]$ (the middle graph, step size = 6) and $[P_1, P_2] = [2^{1023}, 2^{1024} - 1]$ (the graph at the bottom, step size = 7).

Algorithm 5: TrialDivisionRunningTime(P_1, P_2, g)

```
P = SquareRoot( $(P_1 - 1) \cdot (P_2 - 1)$ )  / 2
$F = 1$
$sizeList = factorList = [\,]$
$r$ = GenerateSizeList($sizeList$)
for $i = 0$ to $r$ do
    $Q$ = Exponentiate( $P$, $sizeList[i]$ )
    $factorList[i]$ = RandomPrime( $Q/c_{int}$, $Q \cdot c_{int}$ )
    $F = F \cdot factorList[i]$
end
$I_1 = (P_1 - 1)/(2 \cdot F)$
$I_2 = (P_2 - 1)/(2 \cdot F)$
$n = 2 \cdot$ RandomInt( $I_1$, $I_2$ ) $\cdot F + 1$
$start$ = Clock()
if TrialDivision( $n$ ) then          ▷ TrialDivision with bound $g$
    $a$ = RandomInt( 2, $n - 1$ )
    $success$ = CheckLemma1( $n$, $a$, $factorList$ )
end
$end$ = Clock()
return $end - start$
```

4.4 Implementation of RandomPrime

Our implementation follows the pseudocode of Algorithm 1 with following modifications.

1. If the relative size $1 - \sum_{i=1}^{r} s_i$ is small, it is possible that the interval $[I_1, I_2]$ is too small to contain an R for which $2RF + 1$ is prime. This can cause an infinite loop (lines 20–27 of Algorithm 1), which we prevent by bounding the loop iterations. When the bound is crossed, the prime factors of F are regenerated. If the bound is crossed two times for the same relative sizes $s_1, s_2, \ldots, s_r$, `GenerateSizeList` is also recalled.

   We let the loop iteration bound depend on the size of the interval $[I_1, I_2]$. The bound is crossed as soon as

   $$counter/(128 + \log_2(I_2 - I_1 + 1)) > I_2 - I_1 + 1$$

   holds where *counter* denotes the current number of loop iterations. In this way, we ensure that all possible integers $R \in [I_1, I_2]$ are selected at least once with probability at least $1 - 2^{-128}$ (Coupon Collector Problem [11]). However, we also use $65536 = 2^{16}$ loop iterations as an absolute bound.

2. If the relative size $1 - \sum_{i=1}^{r} s_i$ is small, it is possible that the interval $[\,I_1, I_2\,]$ is too small to contain an integer and thus we can not select an R from the range.

   Our implementation calculates $I'_1 = \lceil I_1 \rceil$ and $I'_2 = \lfloor I_2 \rfloor$ and checks if $I'_1 \leq I'_2$. If this is not fulfilled, the smallest prime factor of F is regenerated and the condition is checked again. If then $I'_1 > I'_2$ still holds all factors of F are regenerated. When all factors of F have to be regenerated a second time, `GenerateSizeList` is recalled as well. As soon as $I'_1 \leq I'_2$, the implementation selects R uniformly at random from the interval $[\,I'_1, I'_2\,]$.

3. The relative size s_i of the prime factor q_i of F, outputted by `GenerateSizeList`, can be arbitrary small. This can cause that there is no prime in the interval $[\,Q_i/c_{int}\,, Q_i \cdot c_{int}\,] = [\,P^{s_i}/c_{int}\,, P^{s_i} \cdot c_{int}\,]$ where c_{int} is a parameter of the algorithm. Therefore, we abort the generation of prime factors of F as soon as $Q_i \cdot c_{int}$ is smaller than 2. Since $s_i \geq s_k$ for all $k > i$, for all subsequent sizes $Q_k, k > i,$ we have $Q_k \cdot c_{int} < 2$ as well.

4. Our implementation of `GenerateSizeList` bounds the length of the returned list to 10 elements. Since the probability of `GenerateSizeList` to output a list of size greater than 10 elements is approximately only 0.5 %, this does not effect the output distribution significantly.

5. Since all prime factors q_i of F are randomly selected from the range $[\,P^{s_i}/c_{int}\,, P^{s_i} \cdot c_{int}\,]$, it is possible that neither $F > \sqrt{n}$ nor $F > R$ (and F is odd) holds. Hence, the implementation always checks if one of these two conditions is fulfilled to correctly apply Lemma 1.

In [6, p. 131], it is suggested to set $P_0 = 10\,000\,000$ and $c_{int} = 1.2$. In our implementation, we use the same c_{int} and set P_0 to the smallest power of 2 greater than 10 000 000, $P_0 = 2^{24}$, so that the decision between base case and recursion case is bit-size dependent. The trial division bounds are set as explained in Section 4.3.

`PrimeTest` function and `TrialDivision` function need a fixed list of small primes. We store these small primes in form of a list of prime gaps starting with the difference 2 between 3 and 5.

$$primeGaps = \{2, 2, 4, 2, 4, \ldots, p_i - p_{i-1}\}$$

The largest reachable prime p_i is chosen so that $p_i^2 \geq P_0$ and p_i is greater than the largest needed trial division bound from Table 4.2. In this way, the primality of all numbers in the base case can be checked and all primes used in `TrialDivision` can be reached.

The source-code of the implementation can be found in Appendix A.

Chapter 5

Running Time Experiments

In this chapter we assess the computational running time of the RANDOMPRIME algorithm implementation, as presented in Chapter 4. We compare the running time of the implementation with the PGA implementation in the OpenSSL library to evaluate the usability of the RANDOMPRIME algorithm in practice. Moreover, we address a modification of the RANDOMPRIME implementation, aimed at reducing the number of regenerations of F. All experiments are performed on Euler [2], a central high-performance cluster of ETH.

5.1 Prime Number Generator in OpenSSL

In OpenSSL version 1.0.2 and version 3.0.0 the function call of the prime generator has the following form:

```
int BN_generate_prime_ex(BIGNUM *ret, int bits, int safe,
const BIGNUM *add, const BIGNUM *rem, BN_GENCB *cb).
```

In OpenSSL 3.0.0 there is also another function `BN_generate_prime_ex2`, which is the same as `BN_generate_prime_ex`, apart from an additional `ctx` parameter which is passed for storing intermediate calculations.

`BN_generate_prime_ex` generates a probable prime p and stores it in `ret`. The outputted p is composite with only negligible error probability. The parameter `bits` denotes the bit length of the generated prime p and the flag `safe` indicates if p needs to be a safe prime, i.e. such that $(p-1)/2$ is also prime. If `add` is set to `NULL`, the two most significant bits of the generated prime are set[1]. With `add` and `rem`, further conditions on the outputted p can

[1]The product of two k-bit numbers, in which the two most significant bits are set, is exactly $2k$ bits long. This is a useful property when we need to multiply two primes in a protocol, for example in RSA.

k	v 1.0.2	v 3.0.0	Overhead
256	2.50	1.54	63 %
512	10.63	5.77	84 %
1024	51.14	41.27	24 %

Table 5.1: Expected running time (in ms) of `BN_generate_prime_ex(p, k, 0, add=2, NULL, NULL)` when generating a `k`-bit prime `p` in OpenSSL 1.0.2k-fips and OpenSSL 3.0.0 Alpha 4, derived by averaging the running time of 2^{20} function calls.

k	v 1.0.2	v 3.0.0	Overhead
256	0.025	0.019	32 %
512	0.050	0.034	47 %
1024	0.123	0.112	10 %

Table 5.2: Expected running time (in ms) of `BN_is_prime_fasttest_ex(n, BN_prime_checks, ctx, 1, NULL)` and `BN_check_prime(n, ctx, NULL)` when `n` is a `k`-bit random odd integer, in OpenSSL 1.0.2k-fips and OpenSSL 3.0.0 Alpha 4, respectively, derived by averaging the running time of 2^{20} function calls.

be specified. Additionally, with `cb` a user defined function can be passed to `BN_generate_prime_ex` to obtain a feedback during the process of prime generation. In our tests we call `BN_generate_prime_ex` to obtain a k-bit prime. We use `add` = 2 so that the second most significant bit is not always set[2].

`BN_generate_prime_ex` generates a prime following the procedure similar to the one explained in [5, pp. 9–10]. Firstly, a pseudo-random initial candidate n_0 of the desired bit-size k is generated. Secondly, using trial division up to a k-dependent bound r [9], the function checks if n_0 is composite. During this process, a table of remainders for the trial division of n_0 is generated. If a divisor of n_0 is found, a new subsequent candidate $n_1 = n_0 + 2$ is chosen and again tested using the same trial division bound. Maintaining the table of remainders, trial division on the subsequent candidate can be done efficiently by avoiding fresh divisions. The table is updated for each subsequent candidate by adding 2 to all remainders in the table. This technique is called sieving.

Thirdly, if a candidate n_i passes trial division, the function runs ℓ rounds of the Miller-Rabin test on the candidate n_i. The value for ℓ is determined according to the bit-size k of n_i: if $k > 2048$, $\ell = 128$, and $\ell = 64$ otherwise [9]. In OpenSSL version 1.0.2, less Miller-Rabin test are applied [5, p. 9]. OpenSSL 1.0.2 calls the function `BN_is_prime_fasttest_ex` for this step and version 3.0.0 uses `BN_check_prime`. As soon as one round of Miller-Rabin test fails, i.e. the primality test function reveals n as a composite, the algorithm

[2]If `add` is not `NULL` and `rem` = `NULL`, the generated prime `p` will fulfil `p` $\equiv 1 \pmod{\texttt{add}}$.

k	RandomPrime	OpenSSL	Overhead
256	9.71	1.54	531 %
512	22.02	5.77	282 %
1024	74.20	41.27	80 %
2048	469.55	329.84	42 %

Table 5.3: Expected running time (in ms) of our implementation of RandomPrime and `BN_generate_prime_ex(p, k, 0, add=2, NULL, NULL)` in OpenSSL 3.0.0 Alpha 4 when generating a `k`-bit prime `p`, derived by averaging the running time of 2^{20} function calls.

restarts from scratch and chooses a new random initial candidate n_0. If n_i passes all the tests, n_i is retuned as probable prime and the prime generator function ends.

The prime generating function, as well as the primality testing function, is faster in OpenSSL 3.0.0 than in OpenSSL 1.0.2 (Table 5.1 and Table 5.2).

5.2 Experimental Results

We compare the running time of our implementation described in Chapter 4 with the running time of the PGA implementation `BN_generate_prime_ex` in OpenSSL as follows. For $k \in \{256, 512, 1024, 2048\}$, we generate k-bit prime numbers utilizing each implementation separately and measure the associated running time.

Our implementation of RandomPrime is slower than the OpenSSL prime generation function `BN_generate_prime_ex`. The overhead depends on k and is presented in Table 5.3.

5.3 Runtime Modification

In RandomPrime (Algorithm 1), after generating F the value for R is chosen uniformly at random from $[I_1, I_2] = [(P_1 - 1)/2F, (P_2 - 1)/2F]$ so that $n = 2RF + 1$ falls into $[P_1, P_2]$. As mentioned in Section 4.4, it is possible that the interval $[I_1, I_2]$ is too small to contain an R for which $n = 2RF + 1$ is prime or the interval is even too small to contain an integer.

Example If $P_1 = 7100$, $P_2 = 9900$, $s_1 = 0.94$ and $r = 1$, we get $2F \approx 5082$ and thus $(I_1, I_2) \approx (1.40, 1.95)$. Hence, we can not select an integer $R \in [I_1, I_2]$.

In such a case, our implementation chooses a new F by regenerating one or all prime factors of F. Regeneration of F can happen at every recursion level and causes additional recursive calls of RandomPrime, which lead to an increased running time of the PGA implementation. However, it maintains the failure probability equal to 0.

k	RandomPrime	OpenSSL	Overhead
256	2.78	1.54	81 %
512	7.39	5.77	28 %
1024	41.37	41.27	0 %
2048	375.98	329.84	14 %

Table 5.4: Expected running time (in ms) of our implementation of modified RandomPrime and `BN_generate_prime_ex(p, k, 0, add=2, NULL, NULL)` in OpenSSL 3.0.0 Alpha 4 when generating a `k`-bit prime `p`, derived by averaging the running time of 2^{20} function calls.

When generating 256, 512 and 1024 bit long primes with RandomPrime, 4–6 %, 2–4 % and 1–3 % of all function calls contain a regeneration of F (at the first level of recursion), respectively. Here we did not count the regenerations in the recursive RandomPrime function calls.

At the cost of reducing the diversity of reachable primes, the number of such additional recursive function calls can be reduced as follows. We introduce a rejection criteria on the list of relative sizes obtained from `GenerateSizeList` as suggested in [6, p. 134]. If

$$\sum_{i=1}^{r} s_i \geq 1 - \frac{C_1}{\log_2(P) + C_2} \tag{5.1}$$

for $C_1 = 10$, $C_2 = 50$ and P as in Algorithm 1, we reject the relative sizes $s_1, s_2, \ldots, s_r$ and call `GenerateSizeList` again. Inequality (5.1) ensures that the relative size of F is not too large and therefore $[\,I_1, I_2\,]$ has a wider range. The smaller P is, the smaller is the right-hand side of (5.1), and thus, the more likely is it that the list of relative sizes is rejected.

Example For $P_1 = 7100$, $P_2 = 9900$, $s_1 = 0.94$ and $r = 1$ as in the example above, (5.1) is fulfilled and therefore we regenerate the relative sizes even before generating F.

Alongside the implementation of Algorithm 1 from Chapter 4, we also implement a modified version. In our modified implementation we use a rejection criteria slightly different to (5.1) so that no logarithm calculations are necessary. We reject the relative sizes $s_1, s_2, \ldots, s_r$ if

$$\sum_{i=1}^{r} s_i \geq 1 - \frac{C_1}{\text{bitSize}(P) + C_2} \tag{5.2}$$

where bitSize(P) denotes the bit-size of P, $C_1 = 10$ and $C_2 = 50$. Since bitSize$(P) \geq \log_2(P)$, (5.2) implies (5.1). The modified RandomPrime implementation is significantly faster than the unmodified one and comparable to the OpenSSL `BN_generate_prime_ex` function for $k = 1024$ (Table 5.4).

We use the same trial division bounds from Section 4.3 for both of our implementations. Rejection criteria (5.2) ensures that the number of regeneration of F is significantly reduced at the cost of reducing the diversity of reachable primes. Primes p for which $(p-1)/2$ is the product of a small R and a prime or the product of a small R and two primes of similar size, can not be reached [6, p. 134]. The unreachable primes include safe primes.

Chapter 6

Discussion

6.1 Advantages and Disadvantages of RandomPrime

In the following, we discuss advantages and disadvantages of the Maurer's PGA captured with RandomPrime (Algorithm 1). We focus on the instantiation of the Maurer's PGA that we presented in Chapter 4. We compare our implementation with the PGA that is implemented in OpenSSL and presented in Section 5.1.

In situations where (proven) primes of bit-length 1024 are needed and a slight deviation from the uniform distribution can be tolerated, RandomPrime with certain modifications is a suitable choice for a PGA. However, for other bit-lengths, our implementation of RandomPrime is too slow.

Some of the introduced modifications restrict the output of the algorithm, i.e. the number of primes that can be reached. The more we restrict the output with these modifications, the faster in expectation is the obtained algorithm. For example, we observe that if we use a lower loop iteration bound in Modification 1 (Section 4.4), we can decrease the running time of our implementation even more.

An advantage of RandomPrime over the OpenSSL probabilistic PGA is that it always generates provable primes. However, in a probabilistic PGA we can break down the error probability to a negligible value to avoid any practical problems.

In comparison to the OpenSSL PGA, a disadvantage of the RandomPrime algorithm is that RandomPrime is significantly harder to understand, as it involves:

- **Application of Lemma 1.** Several conditions required in Lemma 1 need to be checked so that the primality of a prime n is correctly proven.

- **Generation of a random number *RF*.** It needs to be ensured that RF is a random number from $[\,(P_1-1)/2\,,\,(P_2-1)/2\,]$ and therefore that R and the prime factors of F are generated accordingly. To generate the prime factors of F we first generate its relative sizes utilising the appropriately implemented function `GenerateSizeList`.

- **Trial division.** The application of trial division with appropriately set trial division bound g should lead to a fast detection of a composite candidate n to prevent unnecessary costly `CheckLemma1` calls. Therefore, usage of trial division needs to reduce the PGA running time.

- **Potential infinite loops.** We need to carefully bound the number of loop iterations, to prevent crucial infinite loops.

- **Distribution of generated primes.** The generated primes should be close to uniformly distributed over the set of primes in the given interval $[\,P_1\,,\,P_2\,]$. Several parameters of the algorithm have an influence on this property.

Without a good understanding of the algorithm and of its building blocks, transformation of the algorithm into the code might be complex and results in an incorrect or incomplete implementation. This leads to a reduced capability of the implementation, e.g. to a reduced diversity of generated primes, and therefore to security vulnerabilities in the system that uses it.

We also highlight the following disadvantages of RandomPrime.

- The amount of allocations and deallocations of variables and the recursive nature of the algorithm leads to a higher memory utilization. Hence, other PGAs are preferable for devices with bounded memory resources.

- Regenerations of F can occur at multiple levels of the recursion and therefore cause a significant increase in the running time.

- FastPrime presented in Section 3.2.1 only reaches 10 % of all primes.

- The modified RandomPrime algorithm (Section 5.3) can not be used directly to generate safe primes.

Moreover, to implement RandomPrime, one has to tune significantly more parameters than to implement the OpenSSL PGA: P_0, c_{int}, trial division bound g and the loop iteration bound. When implementing the modified version of RandomPrime, parameters C_1 and C_2 also need to be set. However, once properly set, they do not require many further adjustments.

6.2 Further Modifications

With the usage of Lemma 2, RandomPrime can be modified even further. When utilising Lemma 2, only the factorisation of $\sqrt[3]{n}$ is needed to prove the primality of n. Hence, fewer recursive function calls are needed which can reduce the running time. For this modification the `GenerateSizeList` function and the checks of Modification 5 in Section 4.4 need to be adopted accordingly. Then, it can be examined if both or already one of those two changes has an impact on the running time.

Another modification is the customization of the exponentiation-function in line 13 of Algorithm 1 (RandomPrime). For the purposes within the algorithm, a limited functionality of the function can be tolerated, i.e. there is no need for floating-point results or integers closest to exact resulting values, approximations are sufficient. Hence, the power function of GNU MPFR can be replaced with an exponentiation-function with limited functionality that calculates an approximation of P^s in line 13 of Algorithm 1. Such a customized function could be implemented to run faster and to need less memory than the floating-point function of GNU MPFR.

Chapter 7

Conclusion

For several cryptographic purposes, e.g. when using Diffie-Hellman key exchange or RSA, primes are of considerable importance. Incorrect implementations of PGAs can have serious consequences, e.g. revealing RSA secret keys. There are different indicators that assess the quality of a PGA, such as its time complexity or its running time, its success probability (the probability that the generated numbers are indeed prime) and its statistical properties like the uniformity of the output distribution.

Here we presented the RandomPrime PGA proposed by Maurer in 1995 and its practical implementation. The algorithm takes an interval as an input and outputs a random prime from the interval. The primality of the generated primes is proven by Lemma 1, and so the error probability is equal to zero. Additionally, Maurer argues that the output of RandomPrime is close to uniformly distributed over the set of primes in the specified interval.

We did not find any public cryptographic library that provides an implementation of the RandomPrime algorithm. Therefore, we designed and deployed its practical implementation by ourselves. After the deployment, we compared the expected running time[1] of our implementation with that of the PGA function of the widely used cryptographic library OpenSSL. Because of the required high loop iteration bound (Section 4.4, Modification 1) and the numerous recursive function calls in RandomPrime, our implementation is significantly slower than the OpenSSL `BN_generate_prime_ex` function. If we reduce the number of recursive function calls, we can significantly decrease its expected running time. However, the modified version is in expectation slower than `BN_generate_prime_ex` as well. The proposed modification improves the running time at the cost of reduced diversity of reachable primes.

[1]A theoretical running time analysis can be found in [6, pp. 136–141].

One subject for the further research is to examine more closely the parameters of the modified RandomPrime algorithm and derive a conclusion if they can be fine-tuned even further. Another subject is to examine the deviation of the output distribution of the modified RandomPrime algorithm from uniform distribution and draw a conclusion about its impact on applications of PGA. Moreover, comparisons to PGAs of a broader range of cryptographic libraries can be informative.

Appendix A

randomprime.c

A.1 Installation of OpenSSL 3.0.0 on Euler

After downloading *openssl-3.0.0-alpha4.tar.gz* from [8], uploading it on Euler and creating a new folder (here we use `./openssl`), we run the following commands.

1. `tar -xf openssl-3.0.0-alpha4.tar.gz`
2. `cd openssl-3.0.0-alpha4`
3. `./config --prefix = ../openssl --openssldir = ../openssl shared zlib`
   (here you have to use the absolute path instead of `../openssl`)
4. `make`
5. `make test`
6. `make install`

A.2 Compilation on Euler

With the following commands we compile *randomprime.c* on Euler.

```
env2lmod; module load gmp/6.1.2 mpfr/4.0.1;
```

```
gcc -std=gnu99 -o program -I ./openssl/include -L ./openssl/lib
-Wl,-rpath=./openssl/lib randomprime.c -lcrypto -lmpfr -lgmp
```

A.3 Source Code

```
#include <stdio.h>
#include <time.h>
#include <openssl/bn.h>
```

```
#include <gmp.h>
#include <mpfr.h>

#define NUMBER_OF_PRIMES 1000
#define BIT_LENGTH 1024

static const unsigned char primegap[] =
{
   2, 2, 4, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4, 6, 6, 2, 6, 4, 2, 6, 4, 6, 8, 4, 2,
   4, 2, 4,14, 4, 6, 2,10, 2, 6, 6, 4, 6, 6, 2,10, 2, 4, 2,12,12, 4, 2, 4, 6,
   2,10, 6, 6, 6, 2, 6, 4, 2,10,14, 4, 2, 4,14, 6,10, 2, 4, 6, 8, 6, 6, 4, 6,
   8, 4, 8,10, 2,10, 2, 6, 4, 6, 8, 4, 2, 4,12, 8, 4, 8, 4, 6,12, 2,18, 6,10,
   6, 6, 2, 6,10, 6, 6, 2, 6, 6, 4, 2,12,10, 2, 4, 6, 6, 2,12, 4, 6, 8,10, 8,
  10, 8, 6, 6, 4, 8, 6, 4, 8, 4,14,10,12, 2,10, 2, 4, 2,10,14, 4, 2, 4,14, 4,
   2, 4,20, 4, 8,10, 8, 4, 6, 6,14, 4, 6, 6, 8, 6,12, 4, 6, 2,10, 2, 6,10, 2,
  10, 2, 6,18, 4, 2, 4, 6, 6, 8, 6, 6,22, 2,10, 8,10, 6, 6, 8,12, 4, 6, 6, 2,
   6,12,10,18, 2, 4, 6, 2, 6, 4, 2, 4,12, 2, 6,34, 6, 6, 8,18,10,14, 4, 2, 4,
   6, 8, 4, 2, 6,12,10, 2, 4, 2, 4, 6,12,12, 8,12, 6, 4, 6, 8, 4, 8, 4,14, 4,
   6, 2, 4, 6, 2, 6,10,20, 6, 4, 2,24, 4, 2,10,12, 2,10, 8, 6, 6, 6,18, 6, 4,
   2,12,10,12, 8,16,14, 6, 4, 2, 4, 2,10,12, 6, 6,18, 2,16, 2,22, 6, 8, 6, 4,
   2, 4, 8, 6,10, 2,10,14,10, 6,12, 2, 4, 2,10,12, 2,16, 2, 6, 4, 2,10, 8,18,
  24, 4, 6, 8,16, 2, 4, 8,16, 2, 4, 8, 6, 6, 4,12, 2,22, 6, 2, 6, 4, 6,14, 6,
   4, 2, 6, 4, 6,12, 6, 6,14, 4, 6,12, 8, 6, 4,26,18,10, 8, 4, 6, 2, 6,22,12,
   2,16, 8, 4,12,14,10, 2, 4, 8, 6, 6, 4, 2, 4, 6, 8, 4, 2, 6,10, 2,10, 8, 4,
  14,10,12, 2, 6, 4, 2,16,14, 4, 6, 8, 6, 4,18, 8,10, 6, 6, 8,10,12,14, 4, 6,
   6, 2,28, 2,10, 8, 4,14, 4, 8,12, 6,12, 4, 6,20,10, 2,16,26, 4, 2,12, 6, 4,
  12, 6, 8, 4, 8,22, 2, 4, 2,12,28, 2, 6, 6, 6, 4, 6, 2,12, 4,12, 2,10, 2,16,
   2,16, 6,20,16, 8, 4, 2, 4, 2,22, 8,12, 6,10, 2, 4, 6, 2, 6,10, 2,12,10, 2,
  10,14, 6, 4, 6, 8, 6, 6,16,12, 2, 4,14, 6, 4, 8,10, 8, 6, 6,22, 6, 2,10,14,
   4, 6,18, 2,10,14, 4, 2,10,14, 4, 8,18, 4, 6, 2, 4, 6, 2,12, 4,20,22,12, 2,
   4, 6, 6, 2, 6,22, 2, 6,16, 6,12, 2, 6,12,16, 2, 4, 6,14, 4, 2,18,24,10, 6,
   2,10, 2,10, 2,10, 6, 2,10, 2,10, 6, 8,30,10, 2,10, 8, 6,10,18, 6,12,12, 2,
  18, 6, 4, 6, 6,18, 2,10,14, 6, 4, 2, 4,24, 2,12, 6,16, 8, 6, 6,18,16, 2, 4,
   6, 2, 6, 6,10, 6,12,12,18, 2, 6, 4,18, 8,24, 4, 2, 4, 6, 2,12, 4,14,30,10,
   6,12,14, 6,10,12, 2, 4, 6, 8, 6,10, 2, 4,14, 6, 6, 4, 6, 2,10, 2,16,12, 8,
  18, 4, 6,12, 2, 6, 6, 6,28, 6,14, 4, 8,10, 8,12,18, 4, 2, 4,24,12, 6, 2,16,
   6, 6,14,10,14, 4,30, 6, 6, 6, 8, 6, 4, 2,12, 6, 4, 2, 6,22, 6, 2, 4,18, 2,
   4,12, 2, 6, 4,26, 6, 6, 4, 8,10,32,16, 2, 6, 4, 2, 4, 2,10,14, 6, 4, 8,10,
   6,20, 4, 2, 6,30, 4, 8,10, 6, 6, 8, 6,12, 4, 6, 2, 6, 4, 6, 2,10, 2,16, 6,
  20, 4,12,14,28, 6,20, 4,18, 8, 6, 4, 6,14, 6, 6,10, 2,10,12, 8,10, 2,10, 8,
  12,10,24, 2, 4, 8, 6, 4, 8,18,10, 6, 6, 2, 6,10,12, 2,10, 6, 6, 6, 8, 6,10,
   6, 2, 6, 6, 6,10, 8,24, 6,22, 2,18, 4, 8,10,30, 8,18, 4, 2,10, 6, 2, 6, 4,
  18, 8,12,18,16, 6, 2,12, 6,10, 2,10, 2, 6,10,14, 4,24, 2,16, 2,10, 2,10,20,
   4, 2, 4, 8,16, 6, 6, 2,12,16, 8, 4, 6,30, 2,10, 2, 6, 4, 6, 6, 8, 6, 4,12,
   6, 8,12, 4,14,12,10,24, 6,12, 6, 2,22, 8,18,10, 6,14, 4, 2, 6,10, 8, 6, 4,
   6,30,14,10, 2,12,10, 2,16, 2,18,24,18, 6,16,18, 6, 2,18, 4, 6, 2,10, 8,10,
   6, 6, 8, 4, 6, 2,10, 2,12, 4, 6, 6, 2,12, 4,14,18, 4, 6,20, 4, 8, 6, 4, 8,
   4,14, 6, 4,14,12, 4, 2,30, 4,24, 6, 6,12,12,14, 6, 4, 2, 4,18, 6,12, 8, 6,
   4,12, 2,12,30,16, 2, 6,22,14, 6,10,12, 6, 2, 4, 8,10, 6, 6,24,14, 6, 4, 8,
  12,18,10, 2,10, 2, 4, 6,20, 6, 4,14, 4, 2, 4,14, 6,12,24,10, 6, 8,10, 2,30,
   4, 6, 2,12, 4,14, 6,34,12, 8, 6,10, 2, 4,20,10, 8,16, 2,10,14, 4, 2,12, 6,
  16, 6, 8, 4, 8, 4, 6, 8, 6, 6,12, 6, 4, 6, 6, 8,18, 4,20, 4,12, 2,10, 6, 2,
  10,12, 2, 4,20, 6,30, 6, 4, 8,10,12, 6, 2,28, 2, 6, 4, 2,16,12, 2, 6,10, 8,
  24,12, 6,18, 6, 4,14, 6, 4,12, 8, 6,12, 4, 6,12, 6,12, 2,16,20, 4, 2,10,18,
   8, 4,14, 4, 2, 6,22, 6,14, 6, 6,10, 6, 2,10, 2, 4, 2,22, 2, 4, 6, 6,12, 6,
  14,10,12, 6, 8, 4,36,14,12, 6, 4, 6, 2,12, 6,12,16, 2,10, 8,22, 2,12, 6, 4,
   6,18, 2,12, 6, 4,12, 8, 6,12, 4, 6,12, 6, 2,12,12, 4,14, 6,16, 6, 2,10, 8,
  18, 6,34, 2,28, 2,22, 6, 2,10,12, 2, 6, 4, 8,22, 6, 2,10, 8, 4, 6, 8, 4,12,
  18,12,20, 4, 6, 6, 8, 4, 2,16,12, 2,10, 8,10, 2, 4, 6,14,12,22, 8,28, 2, 4,
  20, 4, 2, 4,14,10,12, 2,12,16, 2,28, 8,22, 8, 4, 6, 6,14, 4, 8,12, 6, 6, 4,
  20, 4,18, 2,12, 6, 4, 6,14,18,10, 8,10,32, 6,10, 6, 6, 2, 6,16, 6, 2,12, 6,
  28, 2,10, 8,16, 6, 8, 6,10,24,20,10, 2,10, 2,12, 4, 6,20, 4, 2,12,18,10, 2,
 10, 2, 4,20,16,26, 4, 8, 6, 4,12, 6, 8,12,12, 6, 4, 8,22, 2,16,14,10, 6,12,
 12,14, 6, 4,20, 4,12, 6, 2, 6, 6,16, 8,22, 2,28, 8, 6, 4,20, 4,12,24,20, 4,
  8,10, 2,16, 2,12,12,34, 2, 4, 6,12, 6, 6, 8, 6, 4, 2, 6,24, 4,20,10, 6, 6,
 14, 4, 6, 6, 2,12, 6,10, 2,10, 6,20, 4,26, 4, 2, 6,22, 2,24, 4, 6, 2, 4, 6,
 24, 6, 8, 4, 2,34, 6, 8,16,12, 2,10, 2,10, 6, 8, 4, 8,12,22, 6,14, 4,26, 4,
  2,12,10, 8, 4, 8,12, 4,14, 6,16, 6, 8, 4, 6, 6, 8, 6,10,12, 2, 6, 6,16, 8,
  6, 6,12,10, 2, 6,18, 4, 6, 6, 6,12,18, 8, 6,10, 8,18, 4,14, 6,18,10, 8,10,
 12, 2, 6,12,12,36, 4, 6, 8, 4, 6, 2, 4,18,12, 6, 8, 6, 6, 4,18, 2, 4, 2,24,
  4, 6, 6,14,30, 6, 4, 6,12, 6,20, 4, 8, 4, 8, 6, 6, 4,30, 2,10,12, 8,10, 8,
 24, 6,12, 4,14, 4, 6, 2,28,14,16, 2,12, 6, 4,20,10, 6, 6, 6, 8,10,12,14,10
};

BIGNUM *bn_temp1;
BIGNUM *bn_temp2;
mpz_t mpz_temp1;
mpz_t mpz_temp2;
mpfr_t mpfr_temp;

void insert(unsigned int *list, unsigned int l, unsigned int v)
{
    int i;
    for (i = l-1; (i >= 0 && list[i] < v); i--) {
        list[i+1] = list[i];
    }
    list[i+1] = v;
    return;
}

int generate_size_list(unsigned int *list)
{
    BN_set_word(bn_temp1, 16385);

    for (unsigned int i = 0; i < 10; i++) {
        BN_rand_range(bn_temp2, bn_temp1);
        BN_sub(bn_temp1, bn_temp1, bn_temp2);

        insert(list, i, BN_get_word(bn_temp2));

        unsigned int sum = 16384;
        for (unsigned int j = 0; j <= i; j++) {
            sum -= list[j];
            if (list[j] > sum) {
                return (j+1);
            }
        }
    }
    return 10;
}

int check_lemma_1(mpz_t n, mpz_t a, mpz_t list[], unsigned int length)
{
    mpz_sub_ui(mpz_temp1, n, 1);
    mpz_powm(mpz_temp2, a, mpz_temp1, n);
    if (mpz_cmp_ui(mpz_temp2, 1) != 0) {
        return 0;
    }

    for (unsigned int i = 0; i < length; i++) {
        mpz_divexact(mpz_temp2, mpz_temp1, list[i]);
        mpz_powm(mpz_temp2, a, mpz_temp2, n);
        mpz_sub_ui(mpz_temp2, mpz_temp2, 1);
        mpz_gcd(mpz_temp2, mpz_temp2, n);
        if (mpz_cmp_ui(mpz_temp2, 1) != 0) {
            return 0;
        }
    }
    return 1;
}

int simple_prime_test(unsigned int p)
{
    if (p % 2 == 0) {
        if (p == 2) {
            return 1;
        }
        return 0;
    }

    unsigned int i = 3;
    unsigned int j = 0;
    while (i*i <= p) {
        if (p % i == 0) {
            return 0;
        }
        i += primegap[j++];
    }
    return 1;
}

int trial_division(mpz_t n)
{
    unsigned long i = 3;
    unsigned int g;
    switch(mpz_sizeinbase(n, 2)) {
        case 0 ... 383: g=292; break;
        case 384 ... 767: g=607; break;
        case 768 ... 1535: g=1520; break;
        default: g=3983; break;
    }
    for (unsigned int j = 0; j < g; j++) {
        if (mpz_divisible_ui_p(n, i) != 0) {
            return 0;
        }
        i += primegap[j];
    }
    return 1;
}

void random_number(mpz_t input, mpz_t output)
{
    char *s1 = mpz_get_str(NULL, 16, input);
    BN_hex2bn(&bn_temp1, s1);
    free(s1);

    while (!BN_rand_range(bn_temp2, bn_temp1)) {}

    char *s2 = BN_bn2hex(bn_temp2);
    mpz_set_str(output, s2, 16);
    OPENSSL_free(s2);
}

void random_prime(mpz_t n, mpz_t p1, mpz_t p2)
{
    if (mpz_sizeinbase(p2, 2) < 25) {
        if (mpz_cmp_ui(p2, 2) == 0) {
            mpz_set_ui(n, 2);
        }
        else {
            unsigned long p3 = mpz_get_ui(p1);
            unsigned long p4 = mpz_get_ui(p2);
            BN_set_word(bn_temp1, (p4 - p3 + 1));
            do {
                while (!BN_rand_range(bn_temp2, bn_temp1)) {}
                p4 = BN_get_word(bn_temp2) + p3;
            } while (!simple_prime_test(p4));
            mpz_set_ui(n, p4);
        }
    }
    else {
        mpz_sub_ui(p1, p1, 1);
        mpz_sub_ui(p2, p2, 1);
        mpz_mul(n, p1, p2);
        mpz_tdiv_q_2exp(n, n, 2);
        mpz_sqrt(n, n);
        mpz_t pfl[10];
        mpz_t f;
        mpz_init_set_ui(f, 1);

        mpz_t i1;
        mpz_init(i1);
        mpz_t i2;
        mpz_init(i2);
        unsigned int sizelist[10];
        unsigned int length = generate_size_list(sizelist);
        mpfr_t p;
        mpfr_init_set_z(p, n, MPFR_RNDN);

        for (unsigned int i = 0; i < length; i++) {
            mpfr_set_d(mpfr_temp, (sizelist[i] / 16384.0), MPFR_RNDN);
            mpfr_pow(mpfr_temp, p, mpfr_temp, MPFR_RNDN);
            mpfr_get_z(mpz_temp1, mpfr_temp, MPFR_RNDN);

            if (mpz_cmp_ui(mpz_temp1, 1) == 0) {
                length = i;
                break;
            }
            mpz_mul_ui(i1, mpz_temp1, 10);
            mpz_mul_ui(i2, mpz_temp1, 12);
            mpz_tdiv_q_ui(i1, i1, 12);
            mpz_tdiv_q_ui(i2, i2, 10);
            mpz_init(pfl[i]);
            random_prime(pfl[i], i1, i2);
            mpz_mul(f, f, pfl[i]);
        }
        unsigned int max_length = length;
        unsigned int counter = 0;

        mpz_mul_2exp(n, f, 1);
        mpz_cdiv_q(mpz_temp1, p1, n);
        mpz_tdiv_q(mpz_temp2, p2, n);

        while (mpz_cmp(mpz_temp2, mpz_temp1) < 0) {
            mpz_divexact(f, f, pfl[length-1]);
            random_prime(pfl[length-1], i1, i2);
            mpz_mul(f, f, pfl[length-1]);
            mpz_mul_2exp(n, f, 1);
            mpz_cdiv_q(mpz_temp1, p1, n);
            mpz_tdiv_q(mpz_temp2, p2, n);
            if (mpz_cmp(mpz_temp2, mpz_temp1) >= 0) {
                break;
            }

            mpz_set_ui(f, 1);
            counter++;
            if (counter > 1) {
                counter = 0;
                length = generate_size_list(sizelist);
            }

            for (unsigned int i = 0; i < length; i++) {
                mpfr_set_d(mpfr_temp, (sizelist[i] / 16384.0), MPFR_RNDN);
                mpfr_pow(mpfr_temp, p, mpfr_temp, MPFR_RNDN);
                mpfr_get_z(mpz_temp1, mpfr_temp, MPFR_RNDN);

                if (mpz_cmp_ui(mpz_temp1, 1) == 0) {
                      length = i;
                      break;
                }
                mpz_mul_ui(i1, mpz_temp1, 10);
                mpz_mul_ui(i2, mpz_temp1, 12);
                mpz_tdiv_q_ui(i1, i1, 12);
                mpz_tdiv_q_ui(i2, i2, 10);

                if (i >= max_length) {
                    mpz_init(pfl[i]);
                    max_length++;
                }
                random_prime(pfl[i], i1, i2);
                mpz_mul(f, f, pfl[i]);
            }
            mpz_mul_2exp(n, f, 1);
            mpz_cdiv_q(mpz_temp1, p1, n);
            mpz_tdiv_q(mpz_temp2, p2, n);
        }

        mpz_set(i1, mpz_temp1);
        mpz_sub(i2, mpz_temp2, i1);
        mpz_add_ui(i2, i2, 1);

        unsigned int success = 0;
        unsigned int counter2 = 0;
        counter = 0;
        mpz_t a;
        mpz_init(a);

        while (!success) {
            if (counter > 65536 ||
               mpz_cmp_ui(i2, counter/(128+mpz_sizeinbase(i2, 2))) < 0 ) {
                counter2++;
                if (counter2 > 1) {
                    counter2 = 0;
                    length = generate_size_list(sizelist);
                }
                counter = 0;
                do {
                    mpz_set_ui(f, 1);
                    counter++;
                    if (counter > 1) {
```

```
                    counter = 0;
                    counter2 = 0;
                    length = generate_size_list(sizelist);
                }
                for (unsigned int i = 0; i < length; i++) {
                    mpfr_set_d(mpfr_temp, (sizelist[i] / 16384.0), MPFR_RNDN);
                    mpfr_pow(mpfr_temp, p, mpfr_temp, MPFR_RNDN);
                    mpfr_get_z(i1, mpfr_temp, MPFR_RNDN);

                    if (mpz_cmp_ui(i1, 1) == 0) {
                        length = i;
                        break;
                    }
                    mpz_set(i2, i1);
                    mpz_mul_ui(i1, i1, 10);
                    mpz_tdiv_q_ui(i1, i1, 12);
                    mpz_mul_ui(i2, i2, 12);
                    mpz_tdiv_q_ui(i2, i2, 10);

                    if (i >= max_length) {
                        mpz_init(pfl[i]);
                        max_length++;
                    }
                    random_prime(pfl[i], i1, i2);
                    mpz_mul(f, f, pfl[i]);
                }
                mpz_mul_2exp(n, f, 1);
                mpz_cdiv_q(i1, p1, n);
                mpz_tdiv_q(i2, p2, n);
            } while (mpz_cmp(i2, i1) < 0);
            mpz_sub(i2, i2, i1);
            mpz_add_ui(i2, i2, 1);
            counter = 0;
        }
        random_number(i2, mpz_temp2);
        mpz_add(mpz_temp2, mpz_temp2, i1);
        mpz_mul(n, mpz_temp2, f);
        mpz_mul_2exp(n, n, 1);
        mpz_add_ui(n, n, 1);

        if (trial_division(n)) {
            mpz_mul(mpz_temp1, f, f);
            if (mpz_cmp(mpz_temp1, n) > 0 ||
               (mpz_tstbit(f, 0) == 1 && mpz_cmp(f, mpz_temp2) > 0)) {
                mpz_sub_ui(a, n, 2);
                random_number(a, a);
                mpz_add_ui(a, a, 2);
                success = check_lemma_1(n, a, pfl, length);
            }
        }
        counter++;
    }

    mpz_add_ui(p1, p1, 1);
    mpz_add_ui(p2, p2, 1);
    for (unsigned int i = 0; i < max_length; i++) {
        mpz_clear(pfl[i]);
    }
    mpz_clear(i1);
    mpz_clear(i2);
    mpz_clear(f);
    mpz_clear(a);
```

```
        mpfr_clear(p);
    }
    return;
}

int main(int argc, char *argv[])
{
    mpfr_set_default_prec(BIT_LENGTH);
    mpz_t p1;
    mpz_t p2;
    mpz_init_set_ui(p1, 1);
    mpz_init_set_ui(p2, 1);
    mpz_t n;
    mpz_init(n);
    mpz_mul_2exp(p1, p1, BIT_LENGTH - 1);
    mpz_mul_2exp(p2, p2, BIT_LENGTH);
    mpz_sub_ui(p2, p2, 1);

    for (unsigned int i = 0; i < NUMBER_OF_PRIMES; i++) {
        bn_temp1 = BN_new();
        bn_temp2 = BN_new();
        mpz_init(mpz_temp1);
        mpz_init(mpz_temp2);
        mpfr_init(mpfr_temp);

        random_prime(n, p1, p2);

        BN_free(bn_temp1);
        BN_free(bn_temp2);
        mpz_clear(mpz_temp1);
        mpz_clear(mpz_temp2);
        mpfr_clear(mpfr_temp);

        char *s = mpz_get_str(NULL, 16, n);
        printf("prime p = %s\n", s);
        free(s);
    }

    mpz_clear(p1);
    mpz_clear(p2);
    mpz_clear(n);
    return 1;
}
```

Bibliography

[1] P. Corn et al. Order of an element. https://brilliant.org/wiki/order-of-an-element/. Accessed September 18, 2020.

[2] Euler. https://scicomp.ethz.ch/wiki/Euler. Accessed September 18, 2020.

[3] D. Joyner, R. Kreminski, and J. Turisco. *Applied Abstract Algebra*. Johns Hopkins University Press, 2004.

[4] D. E. Knuth and L. Trabb Prado. Analysis of a simple factorization algorithm. *Theoretical Computer Science*, 3:321–348, 1976.

[5] J. Massimo and K. G. Paterson. A performant, misuse-resistant api for primality testing, 2020. Accessed September 18, 2020.

[6] U. M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology*, 8:123–155, 1995.

[7] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[8] OpenSSL. https://www.openssl.org/source/. Accessed September 2, 2020.

[9] OpenSSL prime testing and prime generation function. https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.c. Accessed September 2, 2020.

[10] H. Riesel. *Prime Numbers and Computer Methods for Factorization*. Birkhäuser, 2nd edition, 1994.

[11] Wikipedia. Coupon collector's problem. https://en.wikipedia.org/wiki/Coupon_collector's_problem. Accessed September 18, 2020.