Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Prime Generation by Incremental Search: An Experimental Exploration

Bachelor Thesis

Filip Dobrosavljević

August 11, 2021

Advisors: Prof. Dr. Kenny Paterson, Filić Mia

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Public-key cryptosystems have allowed us to browse the web securely for many years. These systems often rely on the use of large prime numbers given their unique number-theoretic properties. With the growing computational power and the widespread use of public-key cryptography protocols reliant on prime numbers in practice, larger and larger primes have to be used such that current systems stay secure. Past research has shown some strengths and weaknesses of several Prime Generation Algorithms (PGAs) [2, 16, 8]. However, there exist no studies that provide an experimental comparison between different PGA implementations outputting primes and safe primes. In this thesis we therefore described, implemented and analyzed nine different PGAs, two of which are implemented in the widely used cryptographic libraries OpenSSL and NSS. We found that several implementations for the generation of primes, including the ones from the OpenSSL and NSS library, are a suitable choice. However, for the special case of safe prime generation we found that less implementations are suitable for practical applications.

Contents

Chapter 1

Introduction

Diffie and Hellman's groundbreaking paper about ≪New Directions in Cryptography≫ acted as a quintessential steppingstone to public-key cryptography. Their work had an undeniable impact on our society by providing a way to securely communicate over the world wide web. Different public-key cryptosystems have been proposed since the publication of Diffie and Hellman's paper, including RSA in 1977.

The RSA-Algorithm, named after their publishers Rivest, Shamir and Adleman, generates public and private keypairs by utilizing the product of two distinct prime numbers p and q. For RSA to maintain some of its security properties we need to assume that the factorization of large numbers, specifically the aforementioned product $n = pq$, is hard. However, with the ever-growing computational power it becomes more and more feasible to brute force current public-key cryptosystems that are not using large enough key sizes.

With the widespread use of public-key cryptosystems that are reliant on large primes, it is unquestionable that there is still a great demand for efficient generation of large prime numbers. In practice many different algorithms for prime generation are employed. Although past research has outlined the strengths and weaknesses of some of these algorithms [2, 16, 8], no effort has been made in trying to offer a direct comparison between said algorithms in an experimental setting. This thesis therefore aims to describe, implement and analyze three different Primality Generation Algorithms (PGAs) using three sieving procedures. The total nine different PGAs, constructed from the combination of the three PGAs and three sieves, are implemented to output primes and safe primes using the Incremental Search (IS) paradigm. However, due to the added complexity in regard to safe prime generation our main focus is set on IS PGAs that output primes.

To compare our nine proposed IS PGAs we conduct runtime experiments for our implementations with internal parameter values that satisfy the requirements imposed by [2]. One such requirement, that is of importance in cryptographic applications, is almost uniform distribution over a given set of primes. The set is commonly chosen as all primes of a specific bit-length whose two most significant bits are 1 whereas the desired bit-length is supplied as an input to the IS PGA. To connect to previous research in regard to assessing the output distribution of various IS PGAs as seen in [2] and [16], we carry out a distribution assessment of our implementations in a limited setting.

This thesis is structured as follows. We start by defining IS PGAs and introduce important concepts of IS PGAs in Chapter 2. Next, in Chapter 3 we present descriptions of the algorithms in detail. In Chapter 4 we present our implementations, followed by a runtime and distribution assessment in Chapter 5. The focus will be set on analyzing the runtime of our implementations whereas the distribution assessment is carried out in a limited matter. Finally, in Chapter 6 we discuss the results obtained from Chapter 5 and state the most suitable implementations for practical applications.

Chapter 2

Background

We begin by explaining the IS paradigm for PGAs in further detail. Next, procedures used by the IS PGAs of our interest are presented and a brief introduction to the parameters of our studied IS PGAs is given.

2.1 Incremental search PGAs

In practice we mostly differentiate between two different types of PGAs, specifically trial and error PGAs and IS PGAs. A trial and error PGA uniformly selects a number n from a given set using a cryptographically strong Random Number Generator (RNG). The set is commonly chosen as the set of all odd k-bit numbers whose two most significant bits are set to 1 whereas k is an input to the PGA. The trial and error PGA then tests for primality. If the test fails, the algorithm chooses another number from the set until a prime has been found. An IS PGA on the other hand, randomly starts at a trial n_0, i.e. the starting candidate, chosen from a given set, usually the set of all odd k-bit numbers whose two most significant bits are set to 1, and tests if n_0 is prime. If n_0 is not found to be prime, n_0 is incremented by some number, usually by 2. This procedure is repeated until a prime has been found or a threshold on the maximal number of tested candidates is reached. It is important to note that primality tests used in PGAs are most often probabilistic, i.e. the output number is not guaranteed to be prime. In that case we refer to the output number as a probable prime.

2.1.1 Requirements of IS PGAs

As explained in [2], IS PGAs must meet two requirements:

- Output is prime with some certainty.

As most primality tests used in PGAs are probabilistic, IS PGAs must give some certainty that the output is indeed prime. Fortunately, if a primality test like the Miller-Rabin primality test is used upper bounds for the probability of an IS PGA generating an incorrect output, i.e. output is not prime, can be calculated. As these upper bounds are dependent on the input and internal parameters' values of an IS PGA, we can derive the internal parameters' values from the input parameter, i.e. the output prime bit-size k, to achieve the wanted level of certainty for the output.

- Almost uniform distribution of the prime output

In cryptographic applications prime numbers must remain unpredictable to the user generating them. It is therefore important that primes are generated almost uniformly from some specified set, e.g. the set of all k-bit primes.

2.2 Primality testing algorithms

Primality Testing Algorithms (PTAs) are used to determine whether a number n is prime. PTAs can be divided into zero false positive or probabilistic primality tests. Zero false positive primality tests return with absolute certainty whether n is prime. On the contrary, probabilistic tests declare the input as definitely composite or probably prime.

The Miller-Rabin (MR) primality test is a probabilistic primality test that is widely employed due to its simplicity and efficiency. As many cryptographic libraries offer efficient implementations of the MR primality test, we focus on IS PGAs utilizing the MR test as their primality test. The concept of the MR primality test is checking whether n is a strong probable prime to a given base a, i.e. test whether one of the following congruence relations hold

1. $a^d \equiv 1 \pmod{n}$;
2. $a^{2^r \cdot d} \equiv -1 \pmod{n}$ for some $0 \leq r \leq s$.

Hence, if a witness a is found for which both congruence relations do not hold we can conclude by contraposition that n is definitely composite. However, no simple way of finding a witness is known. It can be shown that by picking base a at random and running t iterations of the MR test, n is mistakenly declared as a probable prime with a probability at most 4^{-t} [6]. Due to the structure of IS PGAs, we can distinguish between two different types of probabilities that are of importance.

2.3 Error probability

We define the error probability of an IS PGA as the probability that an IS PGA outputs a composite. It can be decreased by increasing the number of Miller-Rabin rounds as long as other parameter values for the IS PGA remain the same. As explained in Section 2.1.1, one of the requirements of IS PGAs is that the output is prime with some certainty. To fulfil this requirement it is necessary to derive upper bounds on the error probability of the IS PGA in question and set its parameters' values such that we satisfy the wanted level of certainty.

One important thing to note is that the Miller-Rabin test never rejects a prime. Concretely, this means that if n is a prime the Miller-Rabin test will always declare n as prime.

2.4 Failure probability

In this thesis we define the failure probability of an IS PGA as the probability that an IS PGA fails to return an output number due to failing to find a probable prime. This can be the case if the algorithm has reached its threshold on the maximal number of tested candidates. In practice this treshold is often expressed as a search interval, i.e. the interval from trial n_0 to $n_0 + L$ where L is the length of the search interval. If a candidate goes over the search interval, the IS PGA returns a failure. By choosing an interval consisting of at least one prime we can guarantee a failure probability of 0. A simple upper bound for prime gaps is Bertrand's postulate which states that for every $n > 1$ a prime in $[n, 2n]$ exists [9].

2.5 Sieving

Sieving in the context of IS PGAs is a procedure that aims to reduce IS PGAs' computational cost. The underlying principle of sieving is to generate candidates that will not waste computational resources on the MR test by for example ruling out obvious composites. One way a sieving procedure can find obvious composites is by testing candidates for divisibility with small known primes. If the sieving procedure finds candidates that are divisible by some prime, it can eliminate these composites from the candidate pool. By finding optimized internal parameters' values where the cost of sieving does not outweigh the cost of applying the MR test instead, we can save computational resources with sieving. By possibly precomputing a list of candidates that pass the sieving procedure the cost of sieving is amortized over all candidates and even more computational resources are saved.

In IS PGAs one can see sieving as a subroutine to retrieve the next candidate n to be tested with the MR test. Sieving therefore changes the way in how we search and determine the next candidate in comparison to IS PGAs without sieving.

2.6 Parameters

Here we introduce the basic parameters that IS PGAs of our interest use. It should be noted that in practice users only have to specify the prime output bit-size k. Other parameter values are initialized such that the implementation yields optimal runtimes or are derived from the input parameter k and other internal parameters to meet the desired error and failure probability bounds. As we want to provide a direct comparison between different IS PGAs, we must create an environment that makes finding the optimal parameter values for minimal runtimes bearable. We therefore include all relevant parameters of an IS PGA as inputs to make the search for the optimal parameter values as simple as possible.

For many cryptographic applications it is important that primes have a guaranteed bit-size. We will denote the prime output bit-size k whereas the output prime will be referred to as n.

Some sieving procedures test candidates for divisibility with known primes to rule out obvious composites. In practice these known primes are usually a large list consisting of the first odd primes. We denote the amount of first odd primes to be used in the sieving procedures as r.

As previously mentioned in Section 2.4, search intervals are used in some IS PGAs to limit the amount of candidates that will be tested. Concretely, if the difference between the current candidate and the trial n_0 is larger than the size of the search interval a failure is returned by the IS PGA. The size of the search interval will be referred to as L.

To fulfil the first requirement of IS PGAs from Section 2.1.1, i.e. output is prime with some certainty, we must, among other things, maintain the error probability q of an IS PGA below the wanted level of certainty.

As explained in Section 2.2, we focus on the MR primality test for our IS PGAs. To denote the number of iterations used in the MR primality test we use t.

A glossary with all common parameters can be found in Table 2.1.

k	output bit-size
n_0	trial, starting candidate picked by RNG
r	amount of first odd primes used in sieving
L	size of search interval
q	error probability
t	number of the MR rounds

Table 2.1: Parameters and their notations for IS PGAs of our interest

Chapter 3

Algorithms and Sieves

This chapter presents the three IS PGAs and the three sieves on which we execute our comparative analysis on in detail. Additionally, we describe the parameter value derivation for each IS PGA.

3.1 Generic probability bounds for IS PGAs

Before going into the details of each IS PGA, we derive and explain upper bounds on error and failure probabilities for our IS PGAs. As a starting point, we use the ones found in [2] when analyzing Algorithm 1, denoted as PrimeInc in [2]. PrimeInc is a simple IS PGA that uses parameters t, L and input k to generate probable primes. The subroutine `is_prime(n, t)` in Algorithm 1 uses the MR primality test to test whether n is prime with t MR rounds.

Let M_k denote the set of odd numbers in the interval $[2^{k-1}, 2^k - 1]$, i.e. the set of odd numbers of bit length k. Let E denote the event that PrimeInc outputs a composite and let $q_{k,t,\frac{L}{2}}$[1] $= Prob[E]$ denote the error probability of the PrimeInc Algorithm with input k and parameters t and L where $\frac{L}{2} = c \cdot k \cdot ln(2)$ for some constant c. $q_{k,t,\frac{L}{2}}$ can then be estimated by using the findings of [2]:

$$q_{k,t,\frac{L}{2}} \leq \frac{L}{2 \cdot ln(2)} \left(0.5 \frac{L}{2 \cdot ln(2)} \sum_{m=3}^{M} P_k(C_m) 2^{-t(m-1)} + 0.7 \cdot 2^{-tM} \right), \tag{3.1}$$

where $M \geq 3$ and $C_m \subset M_k$ is the set of composites for which the probability of passing the MR test is larger than 2^{-m}. For $m \leq 2\sqrt{k-1} - 1$ it holds that

[1]We use $q_{k,t,\frac{L}{2}}$ instead of $q_{k,t,s}$ which is used in the paper to be in line with our parameter notation.

$$P_k(C_m) \leq \frac{8}{3}(\pi^2 - 6)\sum_{j=2}^{m} 2^{m-j-(k-1)/j}. \tag{3.2}$$

To get the best bounds on the error probability, c should be chosen as small as possible. However, choosing smaller c increases the failure probability as we narrow down the size of the search interval. For large k, the failure probability of PrimeInc can be estimated with e^{-2c} [2].

Due to the first IS PGA requirement, found in Section 2.1.1, we aim to satisfy the condition that the error and failure probability are smaller than some upper bound, i.e. $\leq 2^{-\gamma}$, $\gamma \in \mathbb{R}_+$. As seen in this section, the error and failure probability are dependent, inter alia, on parameters t, L and input k. Now, to find parameter values for which the error and failure probability are $\leq 2^{-\gamma}$, we select parameters L and, in some cases, r to our liking and use input k to derive t using the upper bound formulae on the error and failure probability of the IS PGA. However, to optimize runtimes of an IS PGA it is necessary to run a benchmark with varying r and L to determine which parameter pair (L, r) gives the best runtime results.

Algorithm 1: PrimeInc

```
   Input: k
   Parameter: t, L
1  $n_0 \in_R M_k$                          // draw trial uniformly
2  n ← $n_0$
3  Loop
4      if is_prime(n,t) then
5          return n
6      else
7          n ← n + 2
8          if $n \geq n_0 + L$ then
9              return failure
10         end
11     end
```

3.2 Algorithms

3.2.1 NSS PGA

The first IS PGA that we analyze is the one implemented in the Network Security Services (NSS) library. The NSS is a cryptographic open-source project maintainted by, inter alia, Google, Red Hat and AOL [10]. We will refer to this IS PGA as the NSS PGA from now on.

The NSS PGA, described in Algorithm 2, is a round based IS PGA that makes use of an additional parameter U, the maximal number of rounds. One round in the NSS PGA, denoted as `NSS_iter` whose pseudocode can be found in Algorithm 3, is similar to the PrimeInc algorithm but utilizes sieving to retrieve the next candidate. If a round does not find a probable prime within distance L between trial n_0 and current candidate n, it returns a failure and another round is started. If all U rounds have passed without finding a probable prime n, a failure is returned. Otherwise, the probable prime n is returned as soon as one rounds succeeds.

The main peculiarity of the NSS PGA is the introduction of parameter U. U allows for smaller choices for L while still maintaining the failure probability below the desired level.

Several subroutines are called in a round. `rand_num(k)` generates a trial n_0 of k-bit length using a cryptographically strong RNG. In the libraries that we have observed the RNG fixes the first two bits of n_0 to 1. As our three studied IS PGAs are compatible with all of our sieves of interest as is, we use the subroutines `init_sieve(...)` and `sieving_algorithm(...)` to denote the sieving procedures. `init_sieve(...)` initializes the sieving data structure, an array usually needed for the purposes of sieving, which is then passed to its corresponding sieving algorithm, `sieving_algorithm(...)`, that calculates the next candidate and stores it into n. The sieve to use can then be selected by changing both `init_sieve(...)` and `sieving_algorithm(...)` to the desired sieving procedure.

The variable `iterator` in Algorithm 3 is used, inter alia, to save the state of the sieve. For example, `iterator` can be used to iterate through a sieve array. Whenever the sieving algorithm is called within the same context again, the iteration through the sieve array can resume with the use of `iterator`. How `iterator` is used in the case of our sieves can be seen in Section 3.3.

Algorithm 2: NSS PGA

```
  Input: k
  Parameter: t, r, L, U
1 j ← 0
2 ret ← $failure$
3 while $ret \neq success$ and $j < U$ do
4     ret ← NSS_iter(n, k, t, r, L)
5     j ← j+1
6 end
7 return n
```

Algorithm 3: NSS_iter

```
   Input: n, k, t, r, L
1  $n_0 \leftarrow$ rand_num(k) // trial generated from strong crypto RNG
2  sieve $\leftarrow$ init_sieve($n_0$, r, L)
3  iterator $\leftarrow 0$
4  do
5      ret $\leftarrow$ sieving_algorithm(sieve, $n_0$, n, iterator)
6      if ret $\neq$ success or $n - n_0 > L$ or num_bits(n) $\neq k$ then
7          return failure
8      end
9  while !is_prime(n,t)
10 return success
```

Derivation of the parameters' values

Let $q_{k,t,\frac{L}{2},U,r,NSS}$ denote the error probability of the NSS PGA with input k and the corresponding parameters set to t, L, U and r. Moreover, let N_k be the random variable capturing k-bit prime gaps. From the prime number theorem [13] we know that the expected k-bit prime gap $E[N_k]$ is well approximated with $kln(2)$. $q_{k,t,\frac{L}{2},U,r,NSS}$ can then be estimated by using the findings of [2]'s error probability $q_{k,t,\frac{L}{2}}$ in the following manner:

$$\begin{aligned}
q_{k,t,\frac{L}{2},U,r,NSS} &= \sum_{i=1}^{U} P[\text{failure in all rounds before round i}] \cdot P[\text{error in round i}] \\
&\approx q_{k,t,\frac{L}{2}} \sum_{i=1}^{U} P[\text{failure in round 1}] \cdots P[\text{failure in round i-1}] && \text{(i.i.d.)} \\
&\lessapprox q_{k,t,\frac{L}{2}} \sum_{i=1}^{U} (e^{-2c})^{(i-1)} \\
&= q_{k,t,\frac{L}{2}} \sum_{i=1}^{U} (e^{-\frac{2L}{2E[N_k]}})^{(i-1)} \\
&= q_{k,t,\frac{L}{2}} \sum_{i=1}^{U} (e^{-\frac{L}{E[N_k]}})^{(i-1)} \\
&= q_{k,t,\frac{L}{2}} \frac{1 - e^{\frac{-UL}{E[N_k]}}}{1 - e^{\frac{-L}{E[N_k]}}}. && \text{(geometric series)}
\end{aligned} \tag{3.3}$$

This leads us to the following conclusion

$$q_{k,t,\frac{L}{2},NSS} \lessapprox q_{k,t,\frac{L}{2}} \cdot \frac{1-e^{\frac{-LU}{E[N_k]}}}{1-e^{\frac{-L}{E[N_k]}}}. \tag{3.4}$$

Therefore, in maintaining the NSS PGA error probability $\leq 2^{-\gamma}$ it suffices to maintain

$$\frac{L}{2 \cdot ln(2)} \left(0.5 \frac{L}{2 \cdot ln(2)} \sum_{m=3}^{M} P_k(C_m) 2^{-t(m-1)} + 0.7 \cdot 2^{-tM} \right) \cdot \frac{1-e^{\frac{-LU}{kln(2)}}}{1-e^{\frac{-L}{kln(2)}}} \tag{3.5}$$

below or equal to $2^{-\gamma}$.

As NSS rounds are independent, we can approximate the failure probability by exponentiating the failure probability of one NSS round by U as follows

$$\begin{aligned} P[\text{failure in a round}]^U &\lessapprox (e^{-2c})^U \\ &= e^{-\frac{2UL}{2kln(2)}} \\ &= e^{-\frac{UL}{kln(2)}}. \end{aligned} \tag{3.6}$$

Hence, we impose restrictions on both U, the maximal NSS rounds, and L by ensuring that the failure probability of the algorithm is small, i.e. $\leq 2^{-\gamma}$

$$\begin{aligned} e^{-\frac{UL}{kln(2)}} &\leq 2^{-\gamma} \\ -\frac{UL}{kln(2)} &\leq -\gamma ln(2) \\ UL &\geq \gamma k ln^2(2). \end{aligned} \tag{3.7}$$

By using the user input k, fixing L to some constant value and desirable upper bounds on the error and failure probabilities, we can derive t and U using the inequalities (3.4) and (3.7), respectively. In detail, we start deriving t by setting $t = 1$ and continuously check if the minimum of (3.5) in M, $3 \leq M \leq 2\sqrt{k-1} - 1$, is below $2^{-\gamma}$ while incrementing t by 1. The first t that satisfies this condition is returned.

3.2.2 Natural PGA

The next IS PGA we analyze is similar to PrimeInc in the sense that once the trial n_0 is selected it never restarts the algorithm with another starting trial. Due to its simple and intuitive structure we refer to it as the Natural PGA. The pseudocode for the Natural PGA is given in Algorithm 4.

Algorithm 4: Natural PGA

```
Input: k
Parameter: t, r, L
1  $n_0 \leftarrow$ rand_num(k)  // trial generated from strong crypto RNG
2  iterator $\leftarrow$ 0
3  sieve $\leftarrow$ init_sieve($n_0$, r, L)
4  ret $\leftarrow$ sieving_algorithm(sieve, $n_0$, n, iterator)
5  if ret $\neq$ success or num_bits(n) $\neq$ k then
6      return failure
7  end
8  while !is_prime(n,t) do
9      ret $\leftarrow$ sieving_algorithm(sieve, $n_0$, n, iterator)
10     if ret $\neq$ 1 or num_bits(n) $\neq$ k then
11         return failure
12     end
13 end
14 return n
```

Derivation of the parameters' values

Our upper bound on the error probability of PrimeInc in (3.1) grows with larger c. In the case of the Natural PGA, we indefinitely search for a prime in the k-bit interval and possibly have a very large c. Therefore, we approximate the error probability using the approach from [2] for $L = \infty$. We first derive L that satisfies the failure probability being $\leq 2^{-(\gamma+1)}$. We further know that the failure probability can be approximated by e^{-2c}. For $\frac{L}{2} = ckln(2)$, we can reformulate the equation to find L for which the failure probability is $\leq 2^{-(\gamma+1)}$ in the following way

$$\begin{aligned}
e^{-2c} &\leq 2^{-(\gamma+1)} \\
e^{-\frac{2L}{2kln(2)}} &\leq 2^{-(\gamma+1)} \\
e^{-\frac{L}{kln(2)}} &\leq 2^{-(\gamma+1)} \\
-\frac{L}{kln(2)} &\leq -(\gamma+1)ln(2) \\
L &\geq (\gamma+1)kln^2(2).
\end{aligned} \tag{3.8}$$

With the received L and the findings of [2], we can give an approximate bound on the error probability for the case of $L = \infty$:

$$q_{k,t,\infty} \leq q_{k,t,\frac{L}{2}} + e^{-2c}(1 + o(1)). \tag{3.9}$$

Now, t for which $q_{k,t,\frac{L}{2}}$ is sufficiently small to achieve an error probability of $\leq 2^{-(\gamma+1)}$ has to considered as a value for the Natural PGA, given input k.

3.2.3 OpenSSL PGA

The last IS PGA we focus on is the one implemented in the OpenSSL library, a toolkit for the TLS and SSL protocols written in the C programming language [12]. We will therefore call this IS PGA the OpenSSL PGA from now on.

The OpenSSL PGA works in a similar way to the NSS PGA. Unlike the NSS PGA, the number of OpenSSL round calls U are unbounded and each round calls the procedure `sieving_algorithm(...)` only once. In other words, each round generates only one candidate n and restarts if candidate n is not a probable prime. Since $U = \infty$, we omit the usage of U in the context of the OpenSSL PGA hereinafter. The pseudocode for the OpenSSL PGA and `OpenSSL_iter` algorithm, which implements one round within the OpenSSL PGA, are given in Algorithm 5 and 6, respectively.

Algorithm 5: OpenSSL PGA

```
  Input: k
  Parameter: t, r, L
1 ret ← $failure$
2 while $ret = failure$ do
3     ret ← OpenSSL_iter(n, k, t, r, L)
4 end
5 return n
```

Derivation of parameters' values

Let the error probability for the OpenSSL PGA with input k and parameters t, L and r be denoted as $q_{k,t,\frac{L}{2},r,OpenSSL}$. Moreover, let G_k^r denote the random variable capturing the gap of k-bit numbers not divisible by the first r odd primes. As the OpenSSL PGA restarts the round after testing one candidate with `is_prime()`, the expected value for the number of independent OpenSSL iterations to retrieve the first probable prime is $\frac{E[N_k]}{2 \cdot E[G_k^r]}$. The intuition behind this is that when drawing the trial n_0 from the k-bit interval uniformly it should take us $\frac{E[N_k]}{2}$ OpenSSL rounds to find a prime in expectation. The division by 2 comes from the fact that we only consider odd numbers from the k-bit interval. Additionally, the usage of our three studied sieves cuts out a fraction of candidates that are not divisible by the first r odd primes and can be approximated with $E[G_k^r] \approx \prod_{i=0}^{r} 1 + \frac{1}{p_i - 1}$ [7]. In

Algorithm 6: OpenSSL_iter

```
Input: n, k, t, r, L
n_0 ← rand_num(k) // trial generated from strong crypto RNG
sieve ← init_sieve(n_0, r, L)
iterator ← 0
ret ← sieving_algorithm(sieve, n_0, n, iterator)
if ret ≠ success or num_bits(n) ≠ k then
    return failure
end
if is_prime(n) then
    return success
else
    return failure
end
```

summary, the error probability can be approximated by taking the expected number of independent OpenSSL rounds and multiplying it by the error probability of one iteration which now implies:

$$\begin{aligned} q_{k,t,\frac{L}{2},r,OpenSSL} &\approx P[\text{error in one OpenSSL iteration}] \\ &\quad \cdot E[\text{\# OpenSSL iter rounds to find probable prime}] \quad \text{(i.i.d.)} \\ &\lessapprox q_{k,t,\frac{L}{2}} \frac{E[N_k]}{2E[G_k^r]}. \\ &\approx q_{k,t,\frac{L}{2}} \frac{k ln(2)}{2\prod_{i=0}^{r} 1 + \frac{1}{p_i - 1}}. \end{aligned} \tag{3.10}$$

As the OpenSSL PGA does not bound the number of OpenSSL rounds its failure probability is always set to 0. We find ourselves in a similar predicament as for the Natural PGA. In detail, the OpenSSL PGA indefinitely searches for odd candidates that are not divisible by the first r odd primes in the k-bit interval and c possibly grows very large. We therefore must first derive c for which the failure probability is $\leq 2^{-\gamma}$. Only then we derive t for which the error probability is $\leq 2^{-\gamma}$. As a starting point we first inspect the failure probability for the case of an indefinite search, i.e. $\frac{L}{2}$ set to ∞, in the PrimeInc Algorithm. We gather from [2] that the error probability of PrimeInc with $\frac{L}{2} = \infty$, i.e. $q_{k,t,\infty}$, is $\leq p_{k,t,\frac{L}{2}} + e^{-2c}(1 + o(1))$ where $p_{k,t,\frac{L}{2}}$ denotes the error probability of an algorithm that runs PrimeInc with some finite $\frac{L}{2}$ and the second term comes from upper bounding the failure probability for the same algorithm. To examine the case of the OpenSSL PGA, we multiply both sides of the inequality with $\frac{E[N_k]}{2E[G_k^r]}$. As a result we receive the following inequality

$$q_{k,t,\frac{L}{2},r,OpenSSL} \lessapprox q_{k,t,\infty} \frac{E[N_k]}{2 \cdot E[G_k^r]} \leq \left(p_{k,t,\frac{L}{2}} + e^{-2c}(1+o(1))\right) \frac{E[N_k]}{2 \cdot E[G_k^r]}. \tag{3.11}$$

By upper bounding (3.11) with $2^{\gamma} = 2^{-(\gamma+1)} + 2^{-(\gamma+1)}$ and the failure probability with e^{-2c}, i.e. omitting the term $o(1)$, we can now split (3.11) into two separate inequalities that must be satisfied as follows:

$$\begin{aligned} p_{k,t,\frac{L}{2}} \frac{E[N_k]}{2 \cdot E[G_k^r]} &\leq 2^{-(\gamma+1)} \\ log_2(p_{k,t,\frac{L}{2}}) + log_2\left(\frac{E[N_k]}{2 \cdot E[G_k^r]}\right) &\leq -(\gamma+1) \\ (\gamma+1) + log_2\left(\frac{E[N_k]}{2 \cdot E[G_k^r]}\right) &\leq -log_2(p_{k,t,\frac{L}{2}}) \\ (\gamma+1) + log_2\left(\frac{E[N_k]}{2\prod_{i=0}^{r} 1 + \frac{1}{p_i - 1}}\right) &\leq -log_2(q_{k,t,\frac{L}{2}}) \end{aligned} \tag{3.12}$$

and

$$\begin{aligned} e^{-2c} \frac{E[N_k]}{2 \cdot E[G_k^r]} &\leq 2^{-(\gamma+1)} \\ -2c \cdot log_2(e) + log_2\left(\frac{E[N_k]}{2 \cdot E[G_k^r]}\right) &\leq -(\gamma+1) \\ \frac{ln(2)}{2}\left((\gamma+1) + log_2\left(\frac{E[N_k]}{2 \cdot E[G_k^r]}\right)\right) &\leq c. \end{aligned} \tag{3.13}$$

The last step of (3.12) can be justified with the fact that the error probability of an algorithm that runs PrimeInc with some finite $\frac{L}{2}$ is exactly $q_{k,t,\frac{L}{2}}$. If we now consider c that satisfies (3.13), we can now derive appropriate t for which (3.12) is satisfied using L, r and input k. The approach to finding t is similar to the one used for the NSS PGA but with $\frac{E[N_k]}{2 \cdot E[G_k^r]}$ instead of $\frac{1-e^{\frac{-UL}{E[N_k]}}}{1-e^{\frac{-L}{E[N_k]}}}$ and $\gamma + 1$ instead of γ.

3.3 Sieves

3.3.1 NSS sieve

The first sieving procedure that we consider is the one implemented in the NSS library. This sieve will therefore be referred to as the NSS sieve.

The NSS sieve initialization function, given in Algorithm 7, sets up an array of size $\frac{L}{2}$ such that $sieve[j]$ being non-zero indicates that $n_0 + 2 \cdot j$ is composite. In detail, in each iteration of the `for` loop the remainder of the trial n_0 and the i-th odd prime is calculated and saved in a variable named `rem`. If n_0 is divisible by $p_{(i+1)}$, `offset` is set to 0, $p_{(i+1)} - rem$ otherwise. Hence, $n_0 +$ `offset` determines the nearest number to n_0 greater than n_0 that is divisible by the current prime. Moreover, all $n_0 +$ `offset` $+ \sigma \cdot p_{(i+1)}$, $\sigma \in \mathbb{N}_0$, are divisible by $p_{(i+1)}$ and therefore marked as composites in the array as well.

By marking composites as non-zero in the array, we can return candidates that pass the trial division with the first r odd primes by searching for zero entries in the array as seen in Algorithm 8. As pointed out in Section 2.5, we save computational cost by prelabeling candidates that passed the NSS sieve. The concept of the NSS sieve is that it eliminates candidates that are divisible with some of the first r odd primes. Therefore, computational cost is saved since eliminated candidates are never tested with the MR primality test.

Algorithm 7: NSS_sieve_init

```
   Input: $n_0$, r, L
 1 sieve ← alloc(L/2)                              // allocate memory
 2 for $i$ ← 0 to $r$ do
 3     rem ← $n_0$ % prime[i]
 4     if $rem = 0$ then
 5         offset ← 0
 6     else
 7         offset ← prime[i] - rem
 8     end
 9     for $j$ ← $offset$ to $L$ increment by prime[i] do
10         if $j$ % 2 = 0 then
11             sieve[j/2] ← 1
12         end
13     end
14 end
15 return success
```

Algorithm 8: NSS_sieve

```
   Input: sieve, $n_0$, n, iterator
1  if iterator > size(sieve) then
2      return failure
3  end
4  while iterator < size(sieve) and sieve[iterator] $\neq 0$ do
5      iterator $\leftarrow$ iterator+1
6      if iterator $\geq$ size(sieve) then
7          return failure
8      end
9  end
10 n $\leftarrow n_0 + 2\cdot$iterator
11 return n
```

3.3.2 OpenSSL sieve

The second sieving procedure we consider is the one used in the OpenSSL PGA implementation. We will refer to it as the OpenSSL sieve.

The OpenSSL sieve initialization function initializes an array of size r and sets its entries to the remainders of the trial n_0 with the first r odd primes. The pseudocode is given in Algorithm 9.

The candidate n is retrieved by checking divisibility of n_0 + `iterator` with all the first r odd primes. If some prime divides n_0 + `iterator`, we increment `iterator` by two, i.e. a new possible candidate is the next odd number. If n_0 + `iterator` passes the divisibility check, we set n to exactly n_0 + `iterator`. In this way a great portion of composites are eliminated from being a candidate n. Therefore when using the OpenSSL sieve within an IS PGA, we do not have to spend computational resources on the MR test to find them. An example instantiation of the OpenSSL sieve is given in Algorithm 10. In the OpenSSL library implementation the variable `max_difference` is set to the largest possible integer subtracted by the r-th odd prime to prevent an overflow happening in line 4 of Algorithm 10.

Algorithm 9: OpenSSL_sieve_init

```
  Input: $n_0$, r, L
1 sieve $\leftarrow$ alloc(r)                    // allocate memory
2 for $i \leftarrow 0$ to $r$ do
3     sieve[i] $\leftarrow n_0$ % prime[i]
4 end
5 return sieve
```

Algorithm 10: OpenSSL_sieve

```
Input: sieve, $n_0$, n, iterator
max_difference                    // used to prevent an overflow
Loop
    for $i \leftarrow 0$ to $r$ do
        if (sieve[i] + iterator) % prime[i] = 0 then
            iterator ← iterator + 2
            if iterator > max_difference then
                return failure
            end
            goto loop
        end
    end
n ← $n_0$ + iterator
iterator ← iterator + 2
return n
```

3.3.3 Dirichlet sieve

The last sieving procedure we consider relies on Dirichlet's theorem on arithmetic progressions to generate prime candidates n within IS PGAs. Thus, we will refer to it as the Dirichlet sieve hereinafter. Dirichlet's theorem states that given an arithmetic progression of terms $dw + a$ for $w = 1, 2, ...$ the series contains an infinite number of primes for coprime d and a. By generating a starting candidate of form $dw + a$ where d and a are coprime, we can find primes by incrementing w as it is guaranteed that this series contains an infinite number of primes as long as we don't overrun the k-bit or search interval.

One important question to consider is whether the Dirichlet sieve uniformly generates primes. As mentioned, d and a must not have a common factor > 1. The amount of all such a are given by Euler's totient function $\varphi(d)$. The proportion of primes in each of those arithmetic progressions is then asymptotically equal to $\frac{1}{\varphi(d)}$ by the stronger form of Dirichlet's theorem [14]. If d is equal to a prime number q, then each of the $q - 1$ progressions

- $q + 1, 2q + 1, ...$
- $q + 2, 2q + 2, ...$
- ...
- $q + q - 1, 2q + q - 1, ...$

asymptotically contain $\approx \frac{1}{(q-1)}$ of primes. Intuitively, this implies that for any fixed d, choosing a and w randomly from the set of coprimes of d and the natural numbers, respectively, and setting $dw + a$ as the candidate n in an IS PGA of our interest yields an almost uniform distribution of an IS PGA's prime output.

In our instantiation we choose d to be the product of the first r odd primes to ensure that a candidate is coprime to the first r odd primes. We will refer to this product as m_r. We model the Dirichlet sieve initialization function such that n_0 stores m_r. The product m_r will then be used in the Dirichlet sieve to find coprime a, i.e. a for which $gcd(m_r, a) = 1$. Once a coprime a has been found, a random z, previously denoted as w, will be generated satisfying the requirement that $n = z \cdot m_r + a$ lies in the k-bit interval, i.e. $[2^{(k-1)}, 2^k - 1]$. We omit the steps on how such z can be found in Algorithm 12 but will describe them when we discuss our implementation of the Dirichlet sieve in the next chapter. When the Dirichlet sieve is called more than once in a row, we define our algorithm to keep adding m_r to the previous candidate n. As an arithmetic progression carries infinite primes, we are guaranteed to find a probable prime as long as we do not overrun the search interval L as given in the NSS PGA or the k-bit interval. The pseudocodes for the Dirichlet sieve initialization and the Dirichlet sieve function are given in Algorithm 11 and 12.

Algorithm 11: Dirichlet_sieve_init

```
  Input: $n_0$, r, L
1 $n_0 \leftarrow 1$
2 for $i \leftarrow 0$ to $r$ do
3     $n_0 \leftarrow n_0 \cdot$prime[i]
4 end
5 return sieve
```

Algorithm 12: Dirichlet_sieve

```
  Input: sieve, $n_0$, n, iterator
1 if iterator = 0 then
2     n ← z·mr+a    // where gcd(mr,a) = 1 and num_bits(n) = k
3     iterator ← 1
4 else
5     n ← n + $n_0$
6 end
7 return success
```

3.4 Safe prime generation

Safe primes have important applications in some Diffie-Hellman key exchange protocol implementations due to their desirable mathematical properties. A safe prime p is of the form $p = 2q + 1$ where both p and q are prime. q is also referred to as a Sophie Germain prime. In this section we will explain the necessary changes to the sieves and IS PGAs from this section to generate safe primes.

3.4.1 Changes to PGAs and derivation of parameter values

There are two additional things that we need to take care of when generating safe primes in IS PGAs. We first have to test primality with MR for both n and $\frac{n-1}{2}$. Secondly, the trial generation requires that the bit of the trial n_0 at index 1 must be set to 1 as else $\frac{n_0-1}{2}$ will be even, i.e. not a Sophie Germain prime. This also implies that the step size in the incremental search changes. For example, from 2 to 4 in the PrimeInc Algorithm described in Section 3.1.

It is important to note that the bound on the error probability of one round within an IS PGA when moving from outputting a probable prime to outputting a probable safe prime does not increase. However, the failure probabilities are not the same as safe primes have a different density than primes. This means that for IS PGAs with bounded search intervals, e.g. the NSS PGA, we must set the thresholds to larger numbers to accomodate the smaller density. A heuristic estimate for the number of Sophie Germain primes less than n is $\xi(n) = 2 \cdot C \cdot \frac{n}{ln^2(n)}$, where $C = \prod_{p>2} \frac{p(p-2)}{(p-1)^2} \approx 0.660161$ denotes Hardy–Littlewood's twin prime constant [15]. Hence, the number of k-bit safe primes is $\approx \xi(2^{k-1}) - \xi(2^{k-2})$. We can approximate the safe prime gap by dividing the total interval length where the k-bit numbers lay in with the amount of k-bit safe primes and receive $\approx 763,000$ for $k = 1024$ (see Appendix A.3 for detailed calculations). In contrast, the average prime gap for $k = 1024$ is ≈ 710.

Finally, to give an estimate for the failure probability of the safe Natural PGA we use the Markov inequality. Sharper bounds like the Chebyshev's inequality require a direct assumption on the variance which is not known yet. Let X be the the random variable capturing the gaps between safe primes. By previous calculations, we know $E[X] \approx 763,000$. To find the needed L that satisfies our requirement of the failure probability being $\leq 2^{-\gamma}$, we first use the Markov inequality as follows

$$P[X \geq a] \leq \frac{E[X]}{a} \tag{3.14}$$

and search for a such that

$$\begin{aligned}
\frac{E[X]}{a} &\leq 2^{-\gamma} \\
log_2(a) &\geq log_2\left(\frac{E[X]}{2^{-\gamma}}\right). \\
&= log_2(E[X]) - (-\gamma) \cdot log_2(2) \\
&= log_2(E[X]) + \gamma.
\end{aligned} \tag{3.15}$$

For $k = 1024$ and $\gamma = 128$ we get $log_2(a) \approx 147.5$. This might lead us to think that L must be set to $\geq 2^{log_2(a)}$ to maintain the failure probability below $2^{-\gamma}$ for the safe Natural PGA. However, this is only true if $Var[X] \leq E[X]^2$ on which we did not find a resource about.

3.4.2 Safe NSS sieve

The Safe NSS sieve initialization now has to additionally factor in that $\frac{n_0-1}{2}$ must also be prime by the definition of safe primes. We therefore check whether the current prime divides n_0 or $\frac{n_0-1}{2}$. As n_0 and $\frac{n_0-1}{2}$ are odd, we must only consider divisibility for n_0 and $n_0 - 1$. Conveniently, we can reuse the offset for n_0 by incrementing it by 1 and setting all array values accordingly. The NSS sieving algorithm itself only changes in the step size of 4 instead 2 as we are dealing with safe primes. The pseudocode for the safe NSS sieve initialization and the safe NSS sieve are given in Algorithm 13 and 14, respectively.

3.4.3 Safe OpenSSL sieve

The safe OpenSSL sieve initialization is equivalent to the non-safe OpenSSL sieve initialization presented in Algorithm 9. On the other hand, the safe OpenSSL sieve checks whether $n_0 + iterator$ or $n_0 - 1 + iterator$ are divisible by any of the first r odd primes as seen in line 4 of Algorithm 15. This is accomplished by checking whether a candidates updated array values are either 0 or 1, i.e. $n_0 + iterator$ or $n_0 - 1 + iterator$ are divisible by the current prime, respectively. The iterator now gets incremented by 4 instead of 2 as seen in line 13 of Algorithm 15.

3.4.4 Safe Dirichlet sieve

The sieve generation step, i.e. the generation of the product m_r, works the same as in the non-safe Dirichlet sieve (see Algorithm 11). The safe Dirichlet sieve however outputs $n = z \cdot m_r + a$ where both a and $a - 1$ are relatively prime to m_r. We add $gcd(m_r, a - 1)$ as an additional requirement because in this way we ensure that the arithmetic progression of form $\frac{z \cdot m_r + a - 1}{2}$ is satisfied for the Sophie Germain prime as well. As the product m_r is always

Algorithm 13: Safe_NSS_sieve_init

```
Input: $n_0$, r, L
sieve ← alloc(L/2)                              // allocate memory
for $i \leftarrow 0$ to $r$ do
    rem ← $n_0$ % prime[i]
    if rem = 0 then
        offset ← 0
    else
        offset ← prime[i] - rem
    end
    for $j \leftarrow offset$ to $2 \star L$ increment by prime[i] do
        if j % 4 = 0 then
            sieve[j/4] ← 1
        end
    end
    if offset = prime[i]-1 then
        offset ← $-1$                   // Do not skip first occurence
    end
    for $j \leftarrow offset + 1$ to $2 \star L$ increment by prime[i] do
        if j % 4 = 0 then
            sieve[j/4] ← 1
        end
    end
end
return success
```

Algorithm 14: Safe_NSS_sieve

```
Input: sieve, $n_0$, n, iterator
if iterator $\geq$ size(sieve) then
    return failure
end
while iterator $<$ size(sieve) and sieve[iterator] $\neq 0$ do
    iterator ← iterator+1
    if iterator $\geq$ size(sieve) then
        return failure
    end
end
n ← $n_0 + 4 \cdot$iterator
return n
```

Algorithm 15: Safe_OpenSSL_sieve

```
Input: sieve, $n_0$, n, iterator
max_difference                    // used to prevent an overflow
Loop
    for $i \leftarrow 0$ to $r$ do
        if (sieve[i] + iterator) % prime[i] $\leq 1$ then
            iterator $\leftarrow$ iterator + 4
            if iterator > max_difference then
                return failure
            end
            goto loop
        end
    end
n $\leftarrow n_0$ + iterator
iterator $\leftarrow$ iterator + 4
return n
```

odd by construction, it is required that z is even and a is odd to guarantee that $\frac{(n-1)}{2}$ is an integer. Consequently, we must increment n by $2 \cdot m_r$ instead of m_r in all the following calls of the safe Dirichlet sieve. One should keep in mind that m_r is stored in the variable n_0 as previously mentioned in Section 3.3.3. The changes are portrayed in Algorithm 16.

Algorithm 16: Safe_Dirichlet_sieve

```
Input: sieve, $n_0$, n, iterator
if iterator = 0 then
    n $\leftarrow$ z $\cdot$ mr+a    // where gcd(mr,a) = 1, gcd(mr,a-1) = 1,
      num_bits(n) = k, z even and a odd
    iterator $\leftarrow$ 1
else
    n $\leftarrow$ n + 2 $\cdot n_0$
end
return success
```

Chapter 4

Implementation

This chapter describes our implementations of the algorithms and sieving procedures described in Chapter 3. In the first section of this chapter we introduce and explain our chosen setup.

4.1 Setup

The primary part of our implementation is written in the programming language C. Some supplementary programs that are used for plotting and doing precomputations are written in Python. All auxillary functions used in the PGAs such as the primality testing functions, RNGs and big number arithmetic are provided by the OpenSSL library (version 3.0.0 Alpha 12) [11]. We choose the OpenSSL library as it provides cryptographically strong RNGs and an efficient implementation of the MR test. Throughout this work we focus on the case of output bit-size $k = 1024$. The results are easily extended to other bit-sizes as explained in Appendix A.2.

We examine nine different IS PGAs, constructed as a combination from the algorithms and sieves described in Chapter 3. They are as follows:

- The NSS PGA with the NSS sieve,
- The NSS PGA with the OpenSSL sieve,
- The NSS PGA with the Dirichlet sieve,
- The Natural PGA with the NSS sieve,
- The Natural PGA with the OpenSSL sieve,
- The Natural PGA with the Dirichlet sieve,
- The OpenSSL PGA with the NSS sieve,
- The OpenSSL PGA with the OpenSSL sieve,

- The OpenSSL PGA with the Dirichlet sieve.

For each of the nine options, we provide an assessment on the performance and distribution. Likewise, we implement and analyse the safe variants of all nine IS PGAs in an equal manner.

4.1.1 Function prototypes

IS PGAs

We implement all three IS PGAs explained in Section 3.2 with the function prototype portrayed in Listing 4.1. A sieve used in an algorithm can then be selected by supplying the function with the corresponding function pointers to the sieve initialization and sieve function. We choose this setup as it allows code reusage and easier debugging.

Listing 4.1: Function prototype of the IS PGAs

```c
int is_pga_name(BIGNUM *p, int k, int t, int r, int L, int
    (*generate_sieve)(...), int (*sieve_algo)(...), int sieve_sz);
```

The functions returns 1[1] if a probable prime has been successfully generated and stored into input `p`. `p` stores the pointer to a `BIGNUM`, a datastructure used in the OpenSSL library for arithmetic operations on integers of arbitrary size. If the function fails to find a probable prime, it returns 0. -1 is returned whenever internal errors happen, for example when OpenSSL library functions return an error. Input `k` and parameters `t`, `r` and `L` are inputs to the function and follow our parameter notation. We use function pointers to the sieve initialization and sieve functions in `generate_sieve` and `sieve_algo`, respectively, as inputs. The OpenSSL PGA, as explained in Section 3.2.3, reallocates the array of its sieve in every round. Therefore, we decided to move the array allocation of the sieves into `is_pga_name()`. With this change we save time as memory for the array has to be allocated only once. In turn, the sieve array allocation size has to be supplied as an input via `sieve_sz`. The sieve array allocation sizes that have to be used for each sieve are represented in Table 4.1. Note that the same sizes hold true for their respective safe variant implementations.

NSS sieve	$L/2$
OpenSSL sieve	$r-1$
Dirichlet sieve	1

Table 4.1: Sieve array allocation sizes for different sieves

[1]instead of using `success` and `failure` as presented in Chapter 3 we use integers as the return code

Sieves

As function pointers are used to select the sieving procedure for an IS PGA, all sieve initialization and sieve functions must have the same function prototype. The prototype of both functions are given in Listing 4.2.

Listing 4.2: Function prototypes for the `sieve_initialization` and `sieve_algorithm` functions

```
int sieve_name_generate_sieve(unsigned short *sieve, int sieve_sz,
    BIGNUM *n0, int r);

int sieve_name(unsigned short *sieve, int sieve_sz, BIGNUM *n, BIGNUM
    *n0, int r, unsigned long *it, int k);
```

A sieve initialization function (`sieve_name_generate_sieve()`) returns 1 if the sieve initialization is successful and 0 otherwise. `sieve` is a pointer that points to the allocated datastructure of size `sieve_sz` whereas `n0` points to a `BIGNUM` storing the trial `n0`. `n0` is initialized in the caller's function `is_pga_name()` using a cryptographically strong RNG. Keep in mind that in the Dirichlet and safe Dirichlet sieve `n0` stores the product m_r. Parameter `r` follows from our parameter notation.

A sieve function, `sieve_name()`, returns 1 if a candidate has been successfully selected and stored in `n`. If the function call fails to find a candidate, 0 is returned. If internal errors occur, the function returns -1. Trial `n0`, input `k` and parameter `r` are defined as before. Lastly, `it` points to an unsigned long that we denoted as `iterator` in our pseudocode notation of Section 3.2.

Safe variants

The safe variants use the same function prototypes as the non-safe variants. We alter the function names by adding the prefix `safe_`.

4.2 Optimal parameter selection

As mentioned in the previous chapters, we must guarantee that both the error and failure probability are $\leq 2^{-\gamma}$ for some $\gamma \in \mathbb{R}_+$. Free parameters in our case are r, the number of first odd primes used in sieving, and additionally the size of the search interval L if an algorithm makes use of it. The number of MR-rounds t and the maximal number of the NSS PGA iterations U are then derived from the free parameters such that the desired error and failure probability is satisfied. Concretely, this means if a runtime test is conducted on varying r and L, an auxillary function is called to calculate the optimal, i.e. the smallest, number of MR rounds t to satisfy the error and failure probability being $\leq 2^{-\gamma}$ for a suitable $\gamma \in \mathbb{R}_+$. In the case of the NSS PGA, U is calculated using (3.7). We set the security parameter γ to 128 for $k = 1024$. However, as the RNG used in our implementations fixes the two most significant bits to 1, keeping the error probability below 2^{-128} in reality keeps it below 2^{-127}. An exception for γ is made for the safe OpenSSL PGA where we set $\gamma = 100$. As we will see in the next chapter, this exception will have no negative implications in regard to our runtime assessment for the safe variants.

The optimal parameter selection was implemented using `generate_optimal_pga_params.py`. This program calls functions defined in `pga_params.py` that calculate the optimized t given an input k, L and r by using the derivations[2] presented in Chapter 3. `generate_optimal_pga_params.py` stores the results into an .csv file which will then be used as an input for the performance and distribution assessment. It is important to note that the optimized parameters' retrieval is only done once at the beginning of each batch and does not influence the performance assessment.

4.3 Modifications in our implementations

In this section we will focus on differences between our implementations and the algorithms described in Chapter 3. It should be noted that most of the implementations follow the corresponding algorithms closely, introducing only minor performance optimizations. The source code for the implementations can be found under [4].

4.3.1 Dirichlet sieve

In the first call of the Dirichlet sieve, we have to generate $n = z \cdot m_r + a$ where n is of k-bits and $gcd(m_r, a) = 1$ holds. The steps to generate such n are as follows. We first repeatedly generate a in $[1, m_r - 1]$ with a cryptographically strong RNG until $gcd(m_r, a) = 1$ holds. Next, we use `BN_rand_range(x, interval_bound)` to generate a number x in the interval [0, `interval_bound`). If we set `interval_bound` = $2^k - (2^{k-1} + 1)$, i.e. the k-bit interval length, we can shift x into the interval $[2^{k-1} - a, 2^k - 1 - a)$ by adding $2^{k-1} - a$ to x. If we now retrieve the remainder of $\frac{x}{m_r}$, we can subtract the remainder off of x. By construction it follows that there must exist a z s.t. $x = z \cdot m_r$. By adding a to x, we construct $n = x + a = z \cdot m_r + a$ where $n \in [2^{k-1}, 2^k - 1]$. As we want to be consistent with other implementations, we set `interval_bound` = $2^k - (2^{(k-1)} + 2^{(k-2)} + 1)$ in our non-safe and safe Dirichlet sieve implementations to make sure that the two most significant bits are set to 1.

[2]As previously explained, we execute all our derivations in such a manner that the part including $q_{k,t,\frac{L}{2}}$ is always maintained below the targeted level times 2

4.3.2 Safe NSS PGA

In Section 3.4.1, we saw that we cannot easily derive L for the safe NSS PGA. We therefore use similar L as in the non-safe variant but increase U to infinity. By doing so we do not have a bound on the number of NSS rounds and do not have to worry about the failure probability. The implication on the error probability will be explained when we analyze the runtime plots of the safe NSS PGA in the next chapter.

4.3.3 Non-safe and safe NSS PGA with the Dirichlet sieve

As the Dirichlet sieve initializes n_0 to m_r, the search interval L is overran in the first sieving call. This is because for larger r the product m_r is significantly larger than the largest possible integer that stores L. This makes the use of L in the sense of the search interval size obsolete. Hence, we propose changes to the NSS PGA with the Dirichlet sieve and the safe NSS PGA with the Dirichlet sieve by utilizing L as the number of m_r additions instead. The implementations can be found in `nss_dir_pga.c` and `safe_nss_dir_pga.c` [4]. For both of the implementations we provide a runtime assessment. However, for the safe variant the runtime assessment is only carried out in limited form. This is because an exploration of the effects of the change on the error probability for the safe variant has not been examined in depth and is left for future work.

4.3.4 Safe Natural PGA with the NSS sieve

As we have no proper safe prime gap estimates, the combination of the safe Natural PGA with the NSS sieve requires us to choose a large L. A large L however implies large memory usage and still no guarantees to find a safe prime. We therefore propose the reuse of the sieve once we have checked all candidates in the array of the NSS sieve. If we overrun the array, the array values get reinitialized with the trial n_0 set to the last candidate +4 instead of returning an error. With this modification, we can continuously search for the next candidate and save memory and runtime by using a much smaller L. The implementation can be found in `safe_nat_nss_pga.c` [4]. The reason why we do not use this modification for the non-safe variant of the Natural PGA with the NSS sieve is that we have sharper bounds on L and the memory usage is significantly smaller than what we would have to use in the safe variant.

Chapter 5

Experimental Assessment

In this chapter we present our running time and distribution experiments that are conducted using the implementations from Chapter 4. We primarily focus on analyzing the runtimes of our implementations whereas the distribution assessment is carried out in limited form. In the first two sections, we explain how the performance and distribution assessments will be conducted. Lastly, after presenting our runtime plots we provide a summary table that carries the parameters' value ranges and minimal runtimes for every IS PGA implementation.

5.1 Performance assessment

All performance assessments are carried out using AMD EPYC 7742 CPUs, which are part of Euler, a high-performance cluster service administered by ETH. For each IS PGA and for each set of free parameters, i.e. r and if needed L, we run 64 batches of 8,192 PGA function calls. For each batch we calculate the mean CPU time of one PGA function call. Given k, r, L, optimized parameter values for t and U are derived using the previously explained methods. The code of our performance assessment can be found in `nss_benchmark.c`, `nat_benchmark.c`, `openssl_benchmark.c` [4]. In order to measure the performance of the safe variants, the corresponding non-safe IS PGA functions have to be exchanged with its safe variants.

5.2 Distribution assessment

For each combination of the IS PGAs and sieves we will carry out a limited distribution check. To check the distribution we run 2,240,000 samples on the parameters' values that gave us the best results in the performance assessment. We then plot a histogram that counts the occurences of the most significant byte of the outputted primes with a 5% acceptance band. The 5%

acceptance band is calculated using the upper and lower range $\mu \pm (\tau \cdot \sqrt{\mu})$, respectively. μ denotes the expected number of occurences per bin, i.e. $2,240,000/64 = 35,000$. We divide by $64 = 2^6$ as 6 bits vary in the most significant byte of the prime output. Only 6 bits vary as the RNG fixes the first two bits to 11. $\tau = 1.960$ is the t-value extracted from the t-distribution table using a confidence of 0.95 and infinite degrees of freedom [5]. It should be noted that the distribution check is rather limited as we only extract the most significant byte of the generated primes. Moreover, the number of samples is not large enough to give an elaborate analysis on the distribution of our IS PGAs. The distribution assessment is implemented in `distribution_check.c` [4].

5.3 Non-safe variants

5.3.1 Runtime results

To compare the runtime results of all implementations, we run 64 batches that provide the mean runtime of 8,192 function calls. The mean of those 64 batches are extracted and plotted in either a heatmap plot or a 2D plot. A heatmap plot is used whenever both r and L are varied, i.e. in the NSS PGA. Additionally, we use a heatmap plot whenever we vary the NSS sieve array size, i.e. in the Natural PGA with the NSS sieve and the OpenSSL PGA with the NSS sieve. A 2D plot is provided otherwise, where the maximum and minimum mean runtime of all 8,192 sample batches are illustrated as well.

For the OpenSSL PGA with the OpenSSL sieve implementation, we additionally provide runtime results of the native OpenSSL implementation, i.e. the IS PGA implemented in the OpenSSL library. The native implementation uses $r = 128$ for the case of $k = 1024$ as provided by [1].

Runtime plots

Figure 5.1 shows the benchmark of the NSS PGA with the NSS sieve. As we can see the NSS PGA with the NSS sieve performs well across a large selection of parameters (dark-blue) but is expectedly more intensive in memory consumption for large L as the NSS sieve allocates an array of size $\frac{L}{2}$. Due to compatibility reasons with our other two sieving procedures, we use `unsigned short` as the datatype for the array. However, in practice a bit-array of size $\frac{L}{2}$ can be used for the NSS sieve which reduces memory consumption.

The NSS PGA with the OpenSSL sieve, whose runtime plot can be found in Figure 5.2, performed similarly to the NSS PGA with the NSS sieve but uses much less memory for large L as the OpenSSL sieve allocates an array of size $r - 1$. Due to the similarity of the OpenSSL and NSS sieve, comparable performance results were expected.

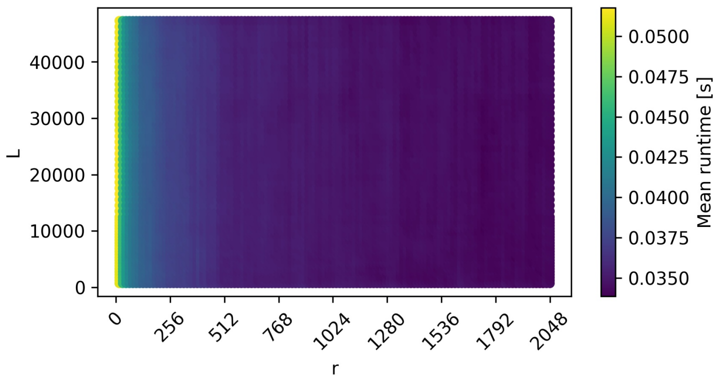


Figure 5.1: The NSS PGA with the NSS sieve

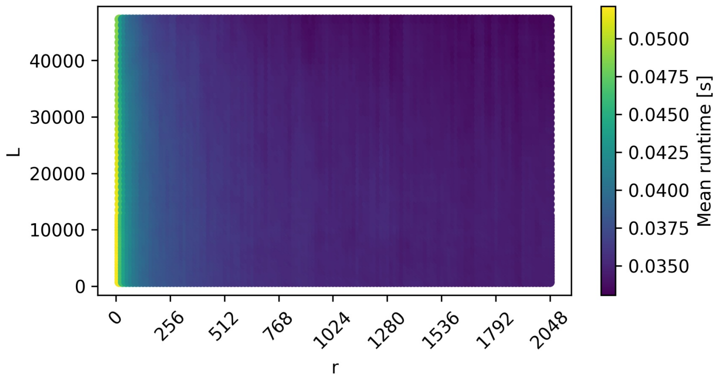


Figure 5.2: The NSS PGA with the OpenSSL sieve

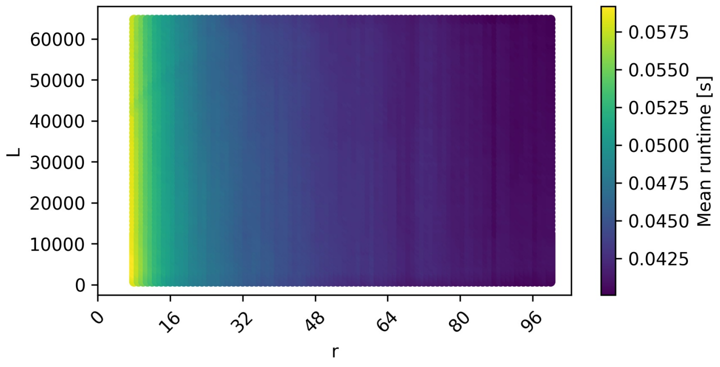


Figure 5.3: The NSS PGA with the Dirichlet sieve

In Figure 5.3 the runtime plot of the NSS PGA with the Dirichlet sieve is depicted. Albeit runtimes seem rather high with minimal runtimes at ≈ 0.0405s in comparison to the NSS PGA with both the NSS and OpenSSL sieve with minimal runtimes at ≤ 0.035s, this implementation requires much less memory due to the nature of the Dirichlet sieve. In contrast, the Dirichlet sieve does not allocate an additional array. However, we observe that this optimization in memory consumption costs runtime. As larger L do not significantly impact our performance results, we think that the main proportion of time is spent for the preparation of the first candidate. After the first candidate has been generated, all following candidates are found in a significantly shorter time as we only have to increment n by m_r. Further work should therefore investigate different strategies for generating the first candidate. Additionally, larger choices for r should be examined. The reason why we did not expand the runtime plots with larger r was due to observed failures. This has to do with the fact that m_r significantly grows with larger r and so does the step size. If m_r grows so large that only a small number of candidates are tested before starting a new round due to overrunning the k-bit interval, we get failures as we set U according to our derivations of Section 3.2.1. The acquired U is in cases of $r > 100$ too small to give any certainty about the output.

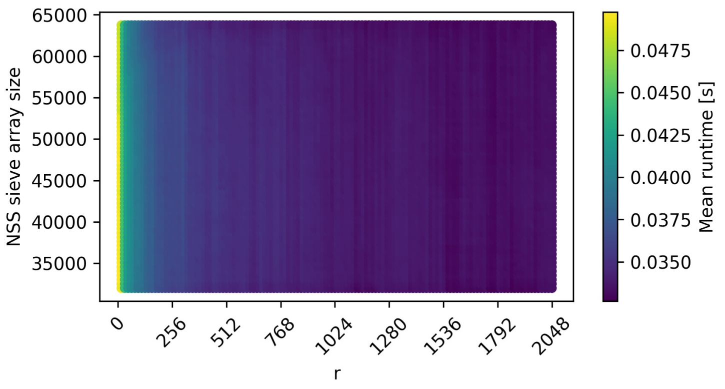


Figure 5.4: The Natural PGA with the NSS sieve

The IS PGA implementation with the fastest runtime was the Natural PGA with the NSS sieve. The runtime plot of the Natural PGA with the NSS sieve in Figure 5.4 indicates that runtimes of $\approx 0.033s$ were achieved as illustrated in the colormap of the heatmap plot. Nevertheless, this implementation also consumed the most memory due to the need of a large NSS sieve array to satisfy the failure probability being $\leq 2^{-128}$ as explained in Section 3.2.2. It should be noted that the difference in memory usage compared to other implementations is not significant for most devices as this implementation consumes only a few more MB. Additionally, the NSS sieve could utilize a bit-array for real world implementations which would lower the memory consumption.

Figure 5.5 illustrates the runtime plot of the Natural PGA with the OpenSSL

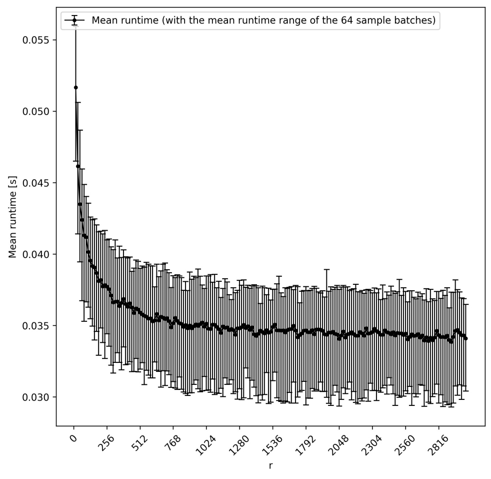


Figure 5.5: The Natural PGA with the OpenSSL sieve

sieve. We observe diminishing returns in terms of performance from using $r \geq 512$. As the OpenSSL sieve allocates an array of size $r - 1$, this implementation does not use as much memory in comparison to the Natural PGA with the NSS sieve and still yields comparable runtime results. Although as before the difference in memory consumption is not significant for most devices. We believe that comparable runtimes to the Natural PGA with the NSS sieves are achieved due to the similarity of both sieves.

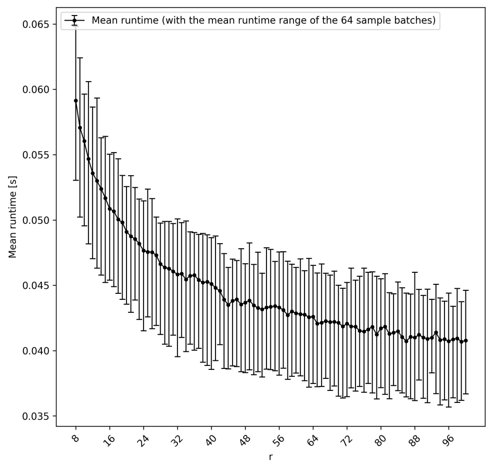


Figure 5.6: The Natural PGA with Dirichlet the sieve

The Natural PGA with the Dirichlet sieve (see Figure 5.6) performs the same as the NSS PGA with the Dirichlet sieve from Figure 5.3. As all three algorithms of Section 3.2 follow a similar structure and do not require the allocation of complex datastructures, a significant portion of the memory usage arises from the sieving procedure. Due to the nature of the Dirichlet sieve, only an `unsigned short` array of size 1 must be allocated, hence why it uses the least amount of memory if we compare the three studied sieves, i.e. the NSS, OpenSSL and Dirichlet sieve. To see all the different array allocation sizes one can examine Table 4.1. Moreover, it can be observed that with larger r better runtimes can be achieved. This trend is also visible in the NSS PGA with Dirichlet sieve combination (see Figure 5.3). The reason for this might be that with larger r the step size m_r is more in line with the distribution of primes.

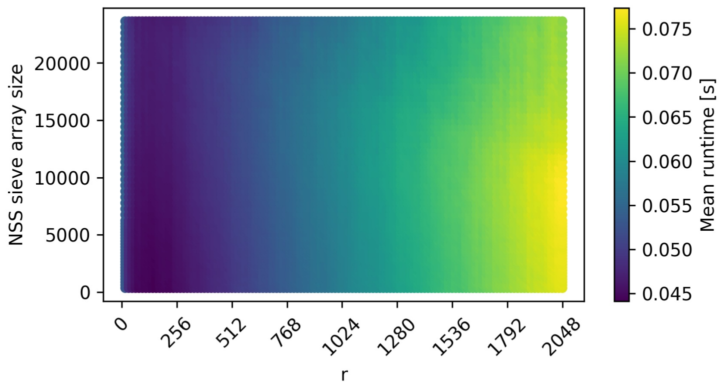


Figure 5.7: The OpenSSL PGA with the NSS sieve

The OpenSSL PGA with the NSS sieve runtime plot in Figure 5.7 suggests that smaller r significantly improve its performance whereas smaller NSS sieve array sizes do not contribute to the performance of the IS PGA. This is contrary to the belief that larger NSS sieve array sizes should worsen performance as the OpenSSL PGA only calls the sieving algorithm once as explained in Section 3.2.3. This suggests that only the first zero entry, i.e. the first candidate not divisible by the first r odd primes, of the array precomputed by the NSS sieve is considered and therefore smaller array sizes should be preferred, especially with the random memory accesses of the NSS sieve. The reason why we don't see a performance dropoff with larger array sizes might be because the given array size is still small enough for the whole array to fit into the cache.

In Figure 5.8 the runtime plot of the OpenSSL PGA with the OpenSSL sieve can be found. Immediately we can see that the cost of the sieve quickly outweighs the gains made by the sieve at around $r = 280$. It should be noted that the OpenSSL PGA implementation with the OpenSSL sieve performed slightly worse than the OpenSSL library implementation. This might have to do with internal optimizations done by OpenSSL, but further investigations would be required to say something meaningful. We leave this out for future

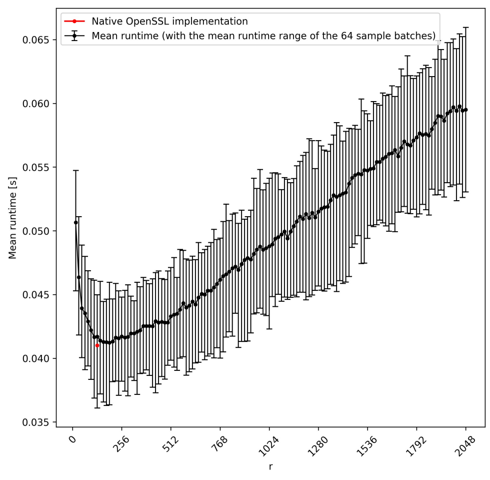


Figure 5.8: The OpenSSL PGA with the OpenSSL sieve

work.

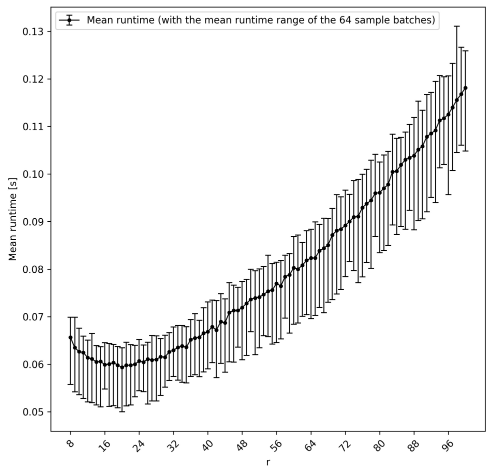


Figure 5.9: The OpenSSL PGA with the Dirichlet sieve

Unsurprisingly, the worst runtimes can be found for the OpenSSL PGA with the Dirichlet sieve as illustrated in Figure 5.9. As an OpenSSL round calls the Dirichlet sieve only once, computational resources are wasted by not incrementing the initially generated n by m_r. Additionally, the product m_r has to be recomputed for every round which increases the computational cost even further. We therefore expect a dip in runtimes for smaller r. This fact can be observed in the curve of Figure 5.9. Future work can optimize the runtimes by precomputing m_r and storing the result in a constant variable before the first OpenSSL round. By doing this the Dirichlet sieve intialization step is no longer needed and computational resources are saved.

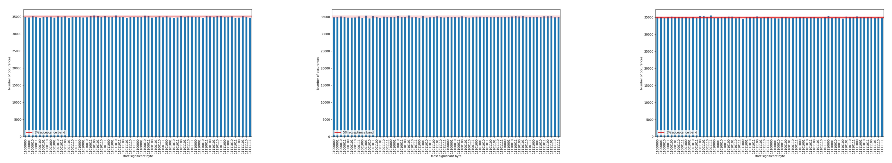
To provide a quick overview of the runtime plots, the parameters' ranges and the minimal runtimes of our implementations are illustrated in Table 5.1.

PGA	Sieve	range for r		range for L		minimal runtime
NSS	NSS	16	2048	700	48000	0.0345s
NSS	OpenSSL	16	2048	700	48000	0.034s
NSS	Dirichlet	8	100	700	64000	0.0405s
Natural	NSS	16	2048	-	-	0.033s
Natural	OpenSSL	16	3024	-	-	0.0345s
Natural	Dirichlet	8	100	-	-	0.042s
OpenSSL	NSS	16	2048	-	-	0.045s
OpenSSL	OpenSSL	16	2048	-	-	0.0415s
OpenSSL	Dirichlet	8	100	-	-	0.06s

Table 5.1: Summary of minimal runtimes and observed ranges for r and L for our non-safe IS PGA implementations

5.3.2 Distribution results

The results of the limited distribution assessment can be found in Figure 5.10, Figure 5.11 and Figure 5.12 for the NSS PGA, Natural PGA and OpenSSL PGA, respectively. Each figure carries three subfigures that illustrate the distribution results of the IS PGA in combination with either the NSS sieve, OpenSSL sieve or Dirichlet sieve. We conclude that the distribution results of all studied non-safe IS PGAs are in the accepted range given our limited assessment in regards to the distribution.



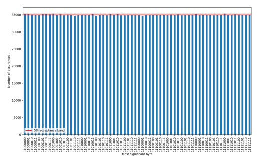
(a) NSS sieve **(b)** OpenSSL sieve **(c)** Dirichlet sieve

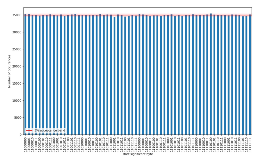
Figure 5.10: Distribution histograms of the NSS PGA in combination with our sieves

5.4 Safe variants

5.4.1 Runtime results

The runtime assessment for the safe variant implementations is carried out equivalently. However, the density and range in where we vary L is sig-


(a) NSS sieve


(b) OpenSSL sieve

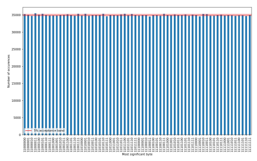
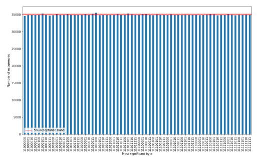
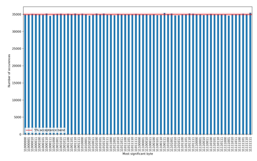

(c) Dirichlet sieve

Figure 5.11: Distribution histograms of the Natural PGA in combination with our sieves


(a) NSS sieve


(b) OpenSSL sieve

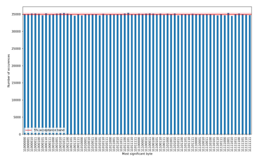

(c) Dirichlet sieve

Figure 5.12: Distribution histograms of the OpenSSL PGA in combination with our sieves

nificantly reduced due to increased runtimes. As seen in Figure 5.9 (c), we reduced the batch size from 8,192 to 2,048 in the benchmark of the safe OpenSSL PGA with Dirichlet sieve as runtimes were substantially larger than those of other combinations.

Like in the case of the non-safe OpenSSL PGA with the OpenSSL sieve, the runtime results of the safe native OpenSSL implementation is provided.

Runtime plots

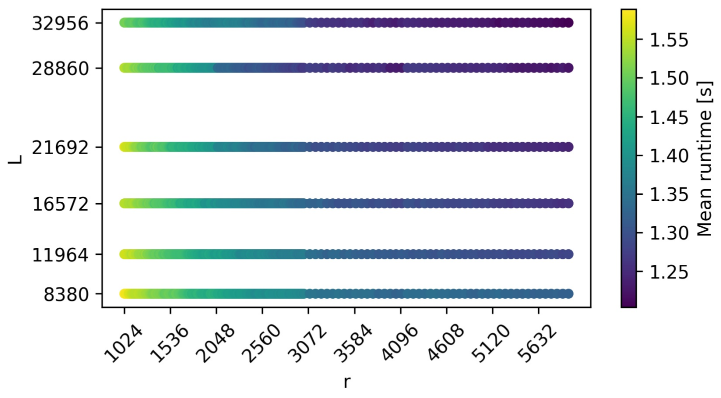


Figure 5.13: The safe NSS PGA with the NSS sieve

Figure 5.13 shows the benchmark plot of the safe NSS PGA with the NSS sieve. Out of all safe IS PGAs this combination performed the best at the expense of using the most memory. The color gradient of the runtime plot indicates that with growing r and L mean runtimes decrease approximately linearly. This is expected as with larger r more composites are weeded out whilst with larger L we amortize the costs of the sieve. However, we believe that much larger choices for r and L do not significantly decrease runtimes which is why we left it out for our runtime plots. Further work should investigate larger choices for r and L to give a conclusive analysis for the safe NSS PGA with the NSS sieve. It is worth mentioning that the optimized parameter derivation is done in the same manner as for the non-safe variant. This implies that the upper bound on the error probability increases by the factor of the expected number of rounds. We have experimentally determined that the expected number of rounds is $\lessapprox 1000$ for $L \geq 8000$. Therefore, instead of setting parameter t while maintaining the error probability $\leq 2^{-\gamma}$, we set optimized values for t that maintain the error probability $\leq 2^{-\gamma + log_2(1000)}$. Therefore, the obtained error probability is $\leq 2^{-\gamma + log_2(1000)}$ which for $\gamma = 128$ and $k = 1024$ is still acceptable. This approach is worthwhile since we expect developers to have a unified procedure for finding parameter t and U for both the safe and non-safe variant. The same holds true for the safe NSS PGA with the OpenSSL sieve. However, the error probability for the safe OpenSSL PGA is left for further work because of its additional complexity.

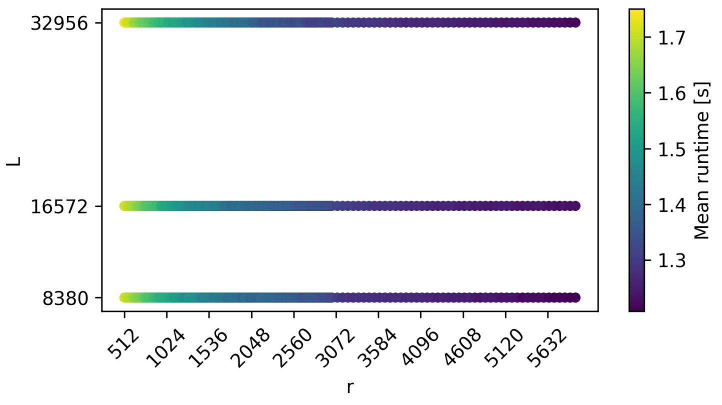


Figure 5.14: The safe NSS PGA with the OpenSSL sieve

In Figure 5.14 the benchmark of the Safe NSS PGA with the OpenSSL sieve is illustrated. We observe similar minimal runtime results to the safe NSS PGA with the NSS sieve (1.25s vs. 1.22s). This combination however uses smaller sieve array sizes which in turn means less memory consumption for a 2.5% increase in minimal runtime. For instance, one can use the parameter configuration $r = 5600$ with $L = 8000$ and yield runtime results of $\approx 1.3s$. In comparison, one would have to use $r = 5600$ with $L = 25000$ in the safe NSS PGA with the NSS sieve for similar runtimes results but would have to utilize as much as 1.5 times the memory for the array allocation. However, as previously mentioned the safe NSS sieve could be implemented only using a bit-array of size $\frac{L}{2}$ which would decrease the amount of memory used. For the OpenSSL sieve it is necessary to use `unsigned short` as the datatype because each entry needs to hold the remainders of the trial, which can be as large as $p_r - 1$. Nevertheless, in practice the difference in overall memory consumption would be negligible as all implementations only differ in a few MB.

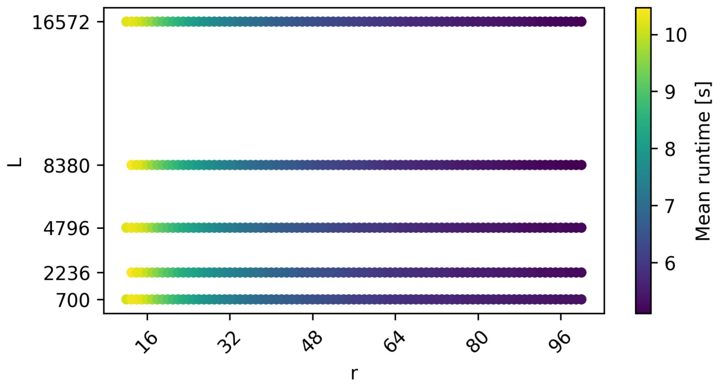


Figure 5.15: The safe NSS PGA with the Dirichlet sieve

The safe NSS PGA with the Dirichlet sieve benchmark results are plotted in Figure 5.15. Alas, this combination produces very poor runtime results. As in the non-safe implementation of this IS PGA (see Figure 5.3) one can observe that different choices for L from the range $[700, 16000]$ do not impact performance that much. Due to the nature of the safe Dirichlet sieve, this indicates that a very large amount of time is spent in generating the first candidate. We believe that finding an appropriate a as explained in Section 4.3.1 might have to do with this. Additionally to the requirement that a must be coprime to m_r in the non-safe Dirichlet sieve, we must now ensure $a - 1$ is coprime to m_r as well. With our approach we repeatedly generate a randomly until both $gcd(m_r, a) = 1$ and $gcd(m_r, a - 1) = 1$. Given this additional requirement, i.e. $gcd(m_r, a - 1) = 1$, we assume that this raises the amount of iterations until such a has been found significantly. Hence, this is why runtimes are substantially larger than for the other two studied sieves. This trend of substantially larger runtimes can be observed in all safe implementations that use the safe Dirichlet sieve, i.e. in Figure 5.15, Figure 5.18 and Figure 5.21.

Similar runtime results to the safe NSS PGA with the NSS sieve can be observed in the safe Natural PGA with the NSS sieve. In Figure 5.16 one can see that larger r reduce the runtimes of the algorithm. Due to our proposed modifications in Section 4.3.4, this implementation now requires much less

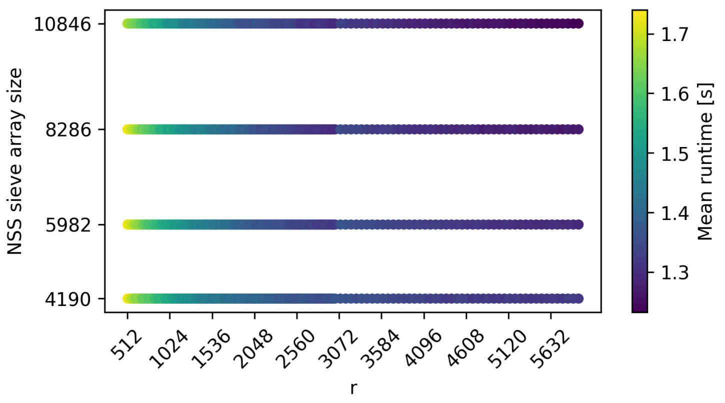


Figure 5.16: The safe Natural PGA with the NSS sieve

memory and performs among the best in our safe implementations with minimal runtimes of ≈ 1.27s.

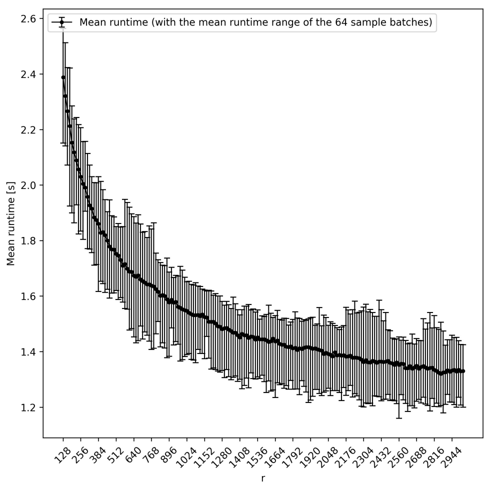


Figure 5.17: The safe Natural PGA with the OpenSSL sieve

The next safe IS PGA implementation is the safe Natural PGA with the OpenSSL sieve whose runtime plot can be seen in Figure 5.17. Immediately one can observe that with larger r the mean runtime stabilizes at $\approx 1.4s$. Therefore, this implementation performs significantly worse than the NSS PGA with the OpenSSL sieve whose minimal runtimes were ≈ 1.25s. Recall that for the non-safe implementations, i.e. for the NSS PGA with the OpenSSL sieve and the Natural PGA with the OpenSSL, minimal runtimes were similar.

In Figure 5.18 the safe Natural PGA with the Dirichlet sieve is depicted. One can see that we achieve similar runtime results to the safe NSS PGA with

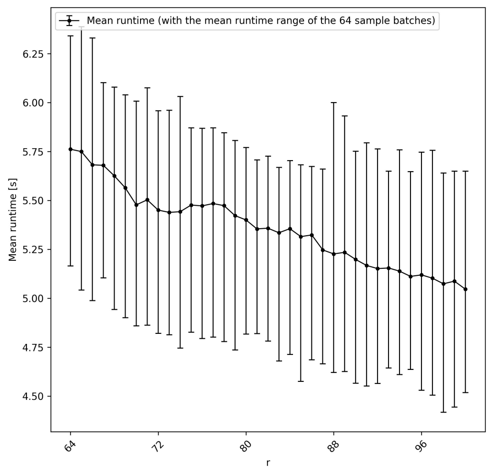


Figure 5.18: The safe Natural PGA with the Dirichlet sieve

the Dirichlet sieve for $r \in [64, 100]$ as seen in Figure 5.15. This comes as no surprise as both the safe Natural PGA and the safe NSS PGA follow a similar structure. The only difference is that the safe NSS PGA restarts the NSS round after L additions of m_r or after overrunning the k-bit interval. However, there does not seem to be a reason to use the safe NSS PGA over the Natural PGA as the safe Natural PGA follows a simpler structure whilst yielding similar runtime results. A simpler structure means easier debugging and less room for error when implementing this algorithm in practice. As with any IS PGA utilizing the Dirichlet sieve, memory consumption is minimized. Nevertheless, for real world applications the runtimes are significantly larger than our best implementations which is why we do not recommend this implementation.

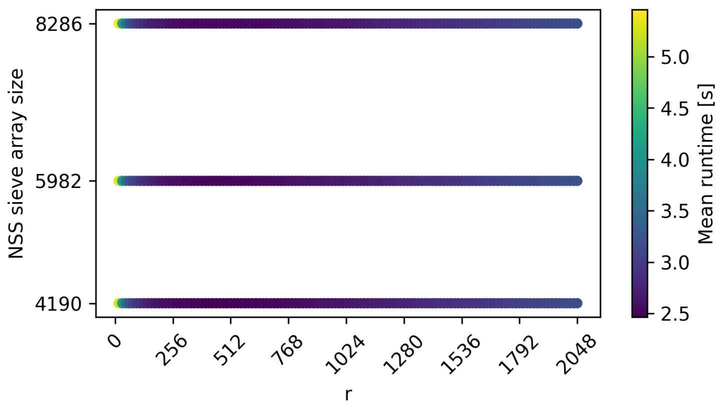


Figure 5.19: The safe OpenSSL PGA with the NSS sieve

Figure 5.19 shows the OpenSSL PGA with the NSS sieve runtime plot. As the safe OpenSSL PGA generates only one candidate n per round, we expect that this safe IS PGA performs worse than other combinations. This is exactly the case. As for the non-safe implementation, we see that the implementation has the best runtimes between $r \approx 128$ and $r \approx 512$ whilst different options of NSS sieve array sizes don't change runtimes much. Therefore smaller NSS sieve array sizes should be preferred in practice in order to save memory. One should keep in mind that the runtime experiments for all safe OpenSSL PGA implementations were conducted using $\gamma = 100$ instead of $\gamma = 128$. An implication of this is that the number of MR rounds t decreases which in turn means lower average runtimes. However, as all safe OpenSSL PGA implementations achieve runtimes significantly worse than our fastest safe IS PGA implementations and given that runtime experiments for our three safe OpenSSL PGAs that satisfy $\gamma = 128$ would only increase runtimes, we can conclude that our safe OpenSSL PGA implementations are not suitable for practical applications even if $\gamma = 128$.

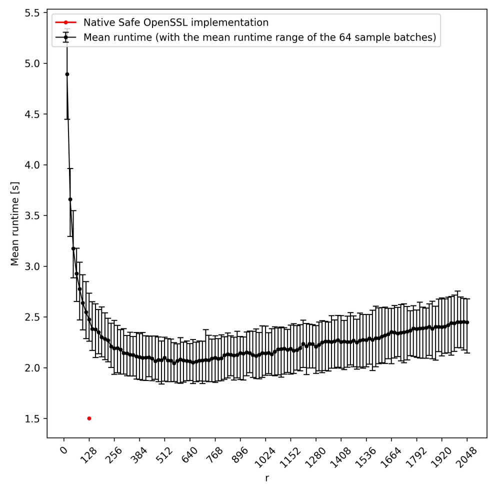


Figure 5.20: The safe OpenSSL PGA with the OpenSSL sieve

The safe OpenSSL PGA with the OpenSSL sieve runtime plot is illustrated in Figure 5.20. Similar to the non-safe variant, this implementation performs worse than the native safe OpenSSL implementation. As previously mentioned this might have to do with internal optimizations done by the OpenSSL library. Interesting to note is that the increase in runtime with higher r is not as significant as in the non-safe implementation. Compared to our other safe implementations this implementation uses slightly less memory at the expense of having higher runtimes.

Again, similar to the case for the non-safe implementations the safe OpenSSL PGA with the Dirichlet sieve (seen in Figure 5.21) has the worst running

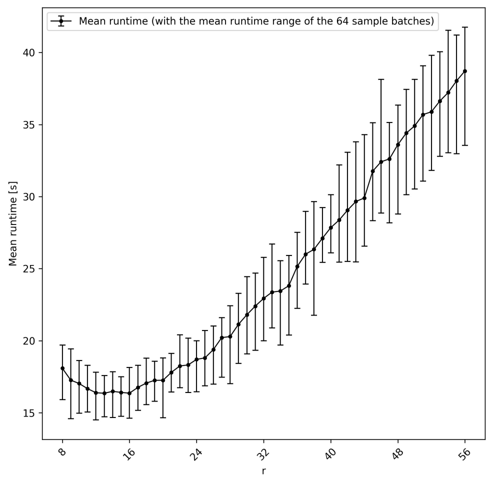


Figure 5.21: The safe OpenSSL PGA with the Dirichlet sieve

times. The reasons for this are equivalent to those for the non-safe variant.

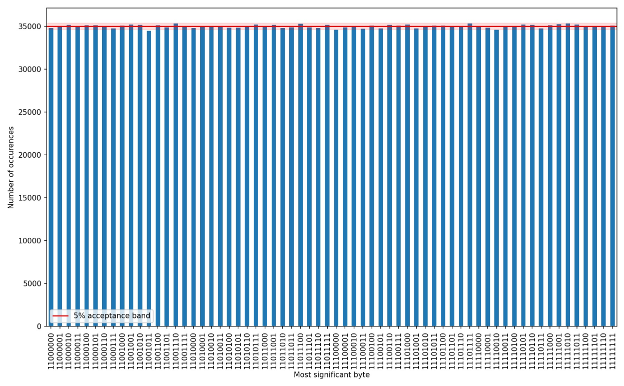
A quick overview of all parameters' ranges and minimal runtimes is summarized in Table 5.2.

PGA	Sieve	range for r		range for L		minimal runtime
NSS	NSS	1024	5968	8380	32956	1.22s
NSS	OpenSSL	512	5968	8380	32956	1.25s
NSS	Dirichlet	16	100	700	16572	5.3s
Natural	NSS	512	5968	-	-	1.27s
Natural	OpenSSL	128	3024	-	-	1.35s
Natural	Dirichlet	64	100	-	-	5.125s
OpenSSL	NSS	16	2048	-	-	2.5s
OpenSSL	OpenSSL	16	2048	-	-	2.1s
OpenSSL	Dirichlet	8	56	-	-	16.5s

Table 5.2: Summary of minimal runtimes and observed ranges for r and L for our safe IS PGA implementations

5.4.2 Distribution results

As the focus of this thesis lies in analyzing IS PGAs in regards to their runtime and memory consumption and due to safe IS PGAs having substantially larger runtimes, we only showcase the distribution results of the fastest implementation, namely the safe NSS PGA with the NSS sieve. We expect other safe variant implementations to yield similar distribution results to Figure 5.22.



(a) Safe NSS PGA with NSS sieve

Figure 5.22: Distribution of the Safe NSS with the NSS sieve

Chapter 6

Discussion and Conclusion

In the following we want to summarize the key findings and give interpretations to the results of Chapter 5. We have seen many different implemenations of algorithms that generate primes under the requirements imposed by cryptographic applications from Section 2.1.1. Two of the nine described algorithms find use in the real world and are employed in cryptographic libraries, namely the OpenSSL and NSS library. Although past research has analyzed the strengths and weaknesses of some of our IS PGAs of interest [2, 16, 8], a experimental comparison of their implementations has not been done yet. In this thesis we therefore described, implemented and analyzed three different IS PGAs using three different sieving procedures. To analyze the total nine different IS PGA implementations, we derived its error and failure probabilities dependent on its input and internal parameters' values. By conducting runtime experiments for our IS PGA implementations with internal parameter values that satisfy the requirements of Section 2.1.1, we are given a framework to directly compare different IS PGA implementations. Now, our goal is to find the most efficient implementation among the studied IS PGAs in regards to runtime and memory consumption and give comments on the suitability of our implementations for different scenarios.

6.1 Non-safe variants

The runtime results of our implementations demonstrate that there exist multiple viable options comparable to the IS PGA implementation in the OpenSSL library in terms of runtime and memory consumption. We gather that all of our studied IS PGA implementations but the OpenSSL PGA with the Dirichlet sieve yield minimal runtimes in the range of $[0.033, 0.045]$s and are therefore feasible choices for the efficient generation of large primes. The difference in overall memory consumption for all of our implementations are not significant for most devices. However, if a device requires minimal memory usage either the Natural PGA with the Dirichlet sieve or the NSS PGA with the Dirichlet sieve should be used. For the best runtime results one should resort to the Natural PGA with the NSS sieve. It achieves minimal mean runtimes of ≈ 0.033s.

When comparing the studied sieves, it becomes apparent that the Dirichlet sieve implementations perform slightly worse than implementations with the two other sieves, namely the OpenSSL sieve and the NSS sieve. This mainly has to do with the fact that our IS PGA implementations were not optimized for the usage of the Dirichlet sieve. Additional research needs to be conducted to determine how well the Dirichlet sieve performs in an optimized setting. In other words different strategies for generating the initial trial of form $n = z \cdot m_r + a$ should be investigated. Moreover, the product m_r should be precomputed such that no additional computational resources have to be wasted.

The data shows that the OpenSSL PGA with the NSS sieve and the Dirichlet sieve performs worse than our other implementations. As one round of the OpenSSL PGA calls the sieving algorithm only once after its initialization, the NSS sieve and the Dirichlet sieve cannot amortize their costs. Especially in combination with the Dirichlet sieve the substantially larger runtimes suggest that the Dirichlet sieve works better with IS PGAs that call the sieving algorithm more than once after initializing the sieve.

6.1.1 Recommendations

The conducted comparison of our IS PGA implementations shows that the Natural PGA with the NSS sieve is the most suitable IS PGA to implement in practice. The runtime plot of the Natural PGA with the NSS sieve shows that the best runtime results out of all combinations were achieved using this IS PGA with parameter values $r \approx 2048$ and NSS sieve array sizes of $\approx 33,000$ with a mean runtime of ≈ 0.033s. In comparison the native OpenSSL libary PGA implementation achieved mean runtimes of ≈ 0.041s. When we compare overall memory consumption with other implementations, the Natural PGA with the NSS sieve uses slightly more memory. As previously mentioned in Chapter 5, memory consumption can be reduced by using a bit-array in the NSS sieve instead of the `unsigned short` array that our implementations use. However, the difference in overall memory consumption even without any memory optimizations is not significant for most modern devices. Furthermore, the Natural PGA with the NSS sieve follows a simple and intuitive structure which eliminates a lot of corner cases that have to be considered. This makes this IS PGA appealing to implement in practice.

6.1.2 Limitations and further work

The Dirichlet sieve is a promising algorithm but further optimizations in our IS PGA implementations are needed to unlock its full potential. For example, one can force the product m_r to be computed only once in one IS PGA call. Also, different strategies for generating the first candidate of form $n = z \cdot m_r + a$ should be examined. In particular, a more efficient way for generating coprime a should be examined. Furthermore, the choice of r was constrained due to the lack of data in regards to the maximal choice of r for given k. To avoid failures in the implementations using the Dirichlet sieve for $k = 1024$, $r = 100$ was used as the maximum. We inspected that with larger choices of r it was not guaranteed that a probable prime is found in the k-bit interval. This has to do with the fact that with larger r the product m_r grows significantly and so does the step size for generating new candidates. For example, in combination with the Natural PGA this means that fewer candidates will be tested and therefore the failure probability significantly increases. Future studies should therefore examine a correct range for r for the Dirichlet sieve in more detail.

When investigating the performance of the OpenSSL library PGA implementation, we found that for the case of $k = 1024$ better runtimes can be achieved by using the Natural PGA with the NSS sieve for example. As the OpenSSL library is widely used, further work should investigate whether the same holds true for larger choices of k. Depending on the results, a new implementation for the OpenSSL library using a different IS PGA might be worthwhile.

As the focus of this study was to provide a direct comparison of IS PGA implementations in regards to runtime, only a limited distribution assessment was conducted. Further work should investigate the output distribution of different IS PGA implementations in a more detailed manner.

6.1.3 Conclusion

The data contributes a clearer understanding of which IS PGAs can be used in certain scenarios, for example for limited-memory devices. All of our IS PGA implementations but the OpenSSL PGA with the Dirichlet sieve are considered viable choices but if memory is highly restrained then the Natural PGA with the Dirichlet sieve or the OpenSSL PGA with the OpenSSL sieve provide an excellent choice where both memory consumption and runtime are balanced. For example, the Natural PGA with the Dirichlet sieve using $r \approx 100$ achieves runtimes of 0.042s whereas the OpenSSL PGA with the OpenSSL sieve yields minimal runtimes of 0.042s with $r \approx 200$. If runtimes are more important, one should resort to the Natural PGA with the NSS sieve using $r \approx 2048$ and NSS sieve array sizes of $\approx 33,000$ as the data suggests that the best runtimes (0.033s) were achieved using these parameters.

As our runtime assessment was only carried out using $k = 1024$, future work should expand the runtime assessment to all k values of interest using the approach described in this thesis.

6.2 Safe variants

Similarily to the non-safe variants multiple viable IS PGAs for the efficient generation of large safe primes exist. From the runtime plots we observe that the following implementations are feasible options when considering implementing safe IS PGAs:

- Safe NSS PGA with the NSS sieve (1.22s),
- Safe NSS PGA with the OpenSSL sieve (1.25s),
- Safe Natural PGA with the NSS sieve (1.27s),
- Safe Natural PGA with the OpenSSL sieve (1.37s),

where approximate minimal mean runtimes of the respective implementation are denoted in brackets. Other implementations had significantly larger minimal runtimes and are therefore not recommended to be used in practice.

When comparing the non-safe variants with the safe variants we found that the choice of L for the safe NSS PGA with the NSS sieve affects the runtimes significantly more than for its non-safe counterpart. This indicates that the average safe prime gap is much larger than the average prime gap which is in line with what we discussed in Section 3.4.1. Furthermore, we see that implementations with the safe Dirichlet sieve perform significantly worse than implementations with our other two sieves. As for the non-safe variants this has to do with the lack of optimizations for the safe Dirichlet sieve in our implementations.

6.2.1 Recommendations

The direct comparison between all safe IS PGA implementations shows that the safe NSS PGA with the NSS sieve performs the best with minimal runtimes of 1.22s using $r \approx 5600$ and $L \approx 32000$. Although overall memory usage is slightly above the ones of other implementations, for most modern devices this difference is not significant.

6.2.2 Limitations and further work

As mentioned with the non-safe variants, further work needs to optimize our IS PGA implementations such that the safe Dirichlet sieve can be efficiently used. Optimizations include precomputation of the product m_r and finding the optimal range of r. Moreover, further studies should investigate different strategies for generating a and z for the trial $n = z \cdot m_r + a$.

We have seen that the failure probability changes dramatically for safe primes. Due to this change, more complex derivations to calculate the error probabilites for the safe variants are required, which we did not discuss other than for the safe NSS PGA. Further work should find the best way to unify the error probability for both the safe and non-safe variants from a theoretical and practical point of view. Additionally, a closer investigation of the error probability for the NSS PGA with the Dirichlet sieve needs to be done to state conclusive facts about said implementation.

Like for the case of the non-safe variants, the distribution assessment for safe IS PGA implementations should be expanded. Future work should also assess runtimes for all k values of interest using the approach described in this thesis.

6.2.3 Conclusion

As seen with the implementations of the non-safe variants, one has to trade off increased runtime for less memory consumption for our safe IS PGA implementations. If runtime is more important, one should use the safe NSS PGA with the NSS sieve with $r \approx 5600$ and $L \approx 32000$ which yields minimal runtimes of 1.22s. However, using large L linearly increases the memory consumption. For a tradeoff between runtime and memory consumption both the safe NSS PGA with the OpenSSL sieve with minimal runtimes of 1.25s at $r \approx 5600$ and $L \approx 8000$ and the safe Natural PGA with the NSS sieve with minimal runtimes of 1.25s at $r \approx 5600$ and NSS sieve array sizes of ≈ 4000 are the way to go. As previously stated the overall difference in memory consumption is not significant and therefore the safe NSS PGA with the NSS sieve should be preferred in practice due to its minimal runtime. However, if one wants to unify the non-safe and safe IS PGAs to one single implementation, then one would first have to examine the behaviour of the Natural PGA with the NSS sieve using our modifications applied on the safe Natural PGA with the NSS sieve. If similar runtime were observed in comparison to our current non-safe variant of the Natural PGA with the NSS sieve, then the unified non-safe and safe Natural PGA with the NSS sieve would stand out as the best option as for both implementations we achieve very fast runtimes and optimal memory consumption.

Appendix A

Appendix

A.1 Installation of OpenSSL 3.0.0

In order to compile our code the installation of the OpenSSL library is necessary. For this thesis we used version 3.0.0 Alpha 12 of OpenSSL which can be downloaded here [11]. After creating a new directory (in our case `./openssl`), we run the following commands in the directory of the downloaded archive.

1. `tar -xf openssl-3.0.0-alpha12.tar.gz`
2. `cd openssl-3.0.0-alpha12`
3. `./Configure --prefix=/openssl --openssldir=/openssl shared zlib`
4. `make`
5. `make test`
6. `make install`

A.2 Compilation of the benchmark files

To run benchmarks using our implementations of the IS PGAs, one must first generate the optimized parameter values using `generate_optimal_pga_params.py`. The variables of `generate_optimal_pga_params.py`, which are explained in Table A.1, must be set to the desired configuration. Files containing the optimized parameter values are then generated in the directory `data/optimal_params/nss_pga/`.

To configure the benchmark files, found in [4], we must change the following variables according to the ones set in `generate_optimal_pga_params.py`.

1. set `k`

variable name	explanation
`k`	output bit-size
`r_max`	maximal r to use
`L_max`	maximal L to use
`r_bound`	bound from where to increment `r` with larger step sizes
`L_bound`	bound from where to increment `L` with larger step sizes

Table A.1: Explanations of the configuration variables in `generate_optimal_pga_params.py`

2. set initial `L`
3. set incrementation size of `L` in `L_inc`
4. set path to output file
5. set path to the generated optimized parameter values
6. set the maximal `L` in the while loop condition
7. set the bound where `L`'s incrementation size changes at the end of the while loop
8. set the sieve used according to our explanations in Section 4.1.1

Assuming the OpenSSL library has been installed in the directory `./openssl` as explained in Section A.1, we compile `pga_name_benchmark.c` with the following command.

```
gcc -std=gnu99 -I/openssl/include -L/openssl/lib -Wl,-rpath=/openssl/lib
pga_name_benchmark.c name_pga.c -lcrypto -o executable_name -O3
```

A.3 Estimating the expected safe prime gap for k=1024

To give an approximation on the expected safe prime gap $E[X]$, we use the arbitrary precision calculator found in [3]. First, we calculate the number of k-bit safe primes with $\xi(2^{k-1}) - \xi(2^{k-2})$.

$\xi(2^{k-1}) =$ 2360278414030469604329784551233417011467616550916022816701
36345562138347365704509237943563372293816630735054885346106372415692
37581049733664261514798864931668696553286657051740028587454096524411
44328936040466813824774322375304267513883074562981848728268440897426
55002813353668821292298596178900140851538

$\xi(2^{k-2}) =$ 1182449806965398203438023693463957291284508554278775372545
22675909485436156031223267313945638190857758840959708970023221937206
18994187257592226319231015380884629369984773648008615376728053700383
98167113043291446194329389762757495309001052707098945873912015109960
89736352423949018288917415171147500625333

Now, both of these values are used to acquire

$\xi(2^{k-1}) - \xi(2^{k-2}) =$ 117782860706507140089176085776945972018310799663724744415613669652652911209673285970629617734102958871894095176376083150478486185868624760720351955678495507840671833018834037314132107260428240274616182299717536763044493261254677220488202185588290285435642578746565266460929719803003381181007752640226205.

Now, we divide the k-bit interval length, i.e. $2^k - 2^{k-1}$, by the number of k-bit safe primes to receive an approximation on the expected safe prime gap as follows

$$E[X] = \frac{2^k - 2^{k-1}}{\xi(2^{k-1}) - \xi(2^{k-2})} \approx 763,000. \tag{A.1}$$

Bibliography

[1] bn_prime.c OpenSSL Github repository. openssl/bn_prime.c at master - openssl/openssl. https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.c. Accessed: 2021-08-05.

[2] Jørgen Brandt and Ivan Damgård. On generation of probable primes by incremental search. In *Annual International Cryptology Conference*, pages 358–370. Springer, 1992.

[3] DCode. Big numbers multiplication calculator - online large multiply. https://www.dcode.fr/big-numbers-multiplication. Accessed: 2021-08-05.

[4] Filip Dobrosavljevic. PrimeGenIS, 2021. doi: https://doi.org/10.5281/zenodo.5172766.

[5] SUNY Polytechnic Institute. Critical values of the student's-t distribution. https://www.itl.nist.gov/div898/handbook/eda/section3/eda3672.htm. Accessed: 2021-07-28.

[6] S Ishmukhametov and B Mubarakov. On practical aspects of the miller-rabin primality test. *Lobachevskii Journal of Mathematics*, 34(4):304–312, 2013.

[7] Jake Massimo and Kenneth G Paterson. A performant, misuse-resistant api for primality testing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–210, 2020.

[8] Preda Mihailescu. Fast generation of provable primes using search in arithmetic progressions. In *Annual International Cryptology Conference*, pages 282–293. Springer, 1994.

[9] Pieter Moree. Bertrand's postulate for primes in arithmetical progressions. *Computers & Mathematics with Applications*, 26(5):35–43, 1993.

[10] Mozilla. Overview of nss - mozilla — mdn. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Overview. Accessed: 2021-08-05.

[11] OpenSSL. Openssl - download and source. https://www.openssl.org/source/. Accessed: 2021-08-05.

[12] OpenSSL. Openssl cryptography and ssl/tls toolkit. https://www.openssl.org/. Accessed: 2021-08-05.

[13] Atle Selberg. An elementary proof of the prime-number theorem. *Annals of Mathematics*, pages 305–313, 1949.

[14] Atle Selberg. An elementary proof of the prime-number theorem for arithmetic progressions. *Canadian Journal of Mathematics*, 2:66–78, 1950.

[15] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge university press, 2009.

[16] Petr Švenda, Matúš Nemec, Peter Sekan, Rudolf Kvašňovský, David Formánek, David Komárek, and Vashek Matyáš. The million-key question—investigating the origins of {RSA} public keys. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 893–910, 2016.