JOHANNES KEPLER
UNIVERSITY LINZ

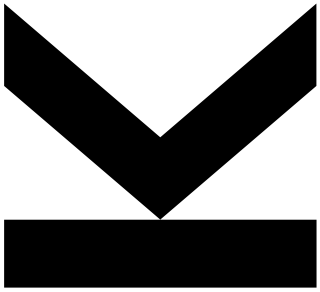
Author
Martin Hengstberger, BSc

Submitted at
Department of Computational Perception

Supervisor
a.Univ.-Prof. DI Dr. Josef Scharinger

September 2016

PROBABILISTIC PRIME NUMBER GENERATION: AN ANDROID IMPLEMENTATION WITH PERFORMANCE EVALUATION



Master's Project
in the Master's Program
Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Contents

Chapter 1

Introduction

The well known cryptosystem RSA by Rivest, Shamir and Adleman (1978) [5] defines an RSA integer n:
"You first compute n as the product of two primes p and q:

$$n = p \cdot q$$

These primes are very large, random primes. Although you will make n public, the factors p and q will be effectively hidden from everyone else due to the enormous difficulty of factoring n."[5]
The primes p and q should be large and distinct. Hence the generation of large primes is needed for the RSA cryptosystem. Similarly other cryptosystems also have a need for large primes e.g. the Digital Signature Algorithm (DSA) or the Diffie-Hellman key exchange (DH). The generation of large primes is a standard task for an encryption library and often done without the users noticing. This project recreates one of the methods for generating such primes on Android.
The sieve of Eratosthenes is typically mentioned in university algorithm courses to find primes. However, there are better ways to do it, one is presented here: the "Miller-Rabin Test".

1.1 Motivation

Large primes are integral to past and present cryptographic systems. Cryptosystems are responsible for confidential online banking, for protecting critical infrastructure e.g. power plants or hospitals, digital signatures that prove the validity of a passport, paying online with a creditcard, Wifi connections or surfing the World Wide Web (https) in a secure manner. Where ever a password is used a cryptosystem is associated with it that likely relies on large primes.
These primes are important because they are part of the mathematical basis of encryption in general. These cryptosystems need primes typically with a length of 1024, 2048 or 3072 Bits. If accidentally an integer believed to be prime were in fact not prime, then it would have a devastating effect on the cryptosystem

(confidentiality of the secret key could be compromised). It could not guarantee its important qualities: Confidentiality, Integrity and Availability anymore.
Primes can only be divided by themselves or 1. Dividing a large integer by all other smaller integers takes a lot of resources. Mathematicians have come up with smarter ways to generate a single large prime, which is here the Miller-Rabin test that can not prove the number to be prime, but the confidence level can be improved arbitrarily by repeating the test.

1.2 Objective

The topic of this project is the generation of large prime numbers in an embedded environment. Resources in embedded environments may be sparse, hence the performance is important (reducing waiting time for the user and increasing battery life). Prime numbers with 1024, 2048 and 3072 Bit shall be generated each 1000 times as a basis for a performance evaluation. The goal of this project is to implement and test a prime number generation routine as well as to conduct a performance evaluation on the implementation. As testing device an emulated device and a real device shall be used and the results shall be compared. The implementation shall further be able to generate small primes to plausibly demonstrate that the generation indeed generates primes. Moreover the implemented method shall be checked with an existing reference implementation from an cryptography library.

1.3 Structure of this Document

After the topic was introduced and motivated, the basic principles of the Miller-Rabin test are explained by discussing the pseudo code and the accuracy of the test. In the next chapter the implementation contains first an overview and description of the process used to generate and test primes. Further more the user interface of the Android app, the actual implementation of the Miller-Rabin test, the systematic of the performance measurement and the reference implementation (crypto library) are discussed. Then a short manual describes how to install and use the app.
Then most importantly the results of the performance evaluation are presented. This document ends with a summary and suggestions for future work.

Chapter 2

Theoretical Principle

There are many ways of generating large primes. Probabilistic and provable methods can be distinguished. Provable constructions are more complicated and slower, but guarantee to return a prime where as probabilistic methods are simpler but only give a certain probability that the candidate is indeed prime. There is the FIPS-186-4 [6] (2013) standard that describes technical requirements of prime generation and its context well. Which served as main source of this project. The Miller-Rabin test was chosen from the standard to be implemented.
This test is based on the "basic principle" that state if $x^2 = y^2 (mod\, n)$ but $x \neq \pm y (mod\, n)$ then n is a composite. That is how n can be discovered not to be prime. However, this is not a proof of primality it could just not be identified as composite.
The general algorithm consists of three steps where n is the prime candidate:
s ... positive integer, counter variable that is incremented as needed
d ... intermediate variable (to be calculated)

1. Find s, d such that $n - 1 = 2^s \cdot d$
2. Choose a random base a that is in the range $[1, n - 2]$
3. Check if $x = a^d (mod\, n) \equiv 1$ or $x \equiv n - 1$ holds

For all primes the check $x \equiv 1$ or $x \equiv n - 1$ holds. If one of the two checks ($x \equiv 1$ or $x \equiv n - 1$) does not hold, then n is proven to be composite. Steps two and three can be repeated with different bases to increase the probability of n being prime. Several iterations with different bases are part of the algorithm.
There is a version of the Miller-Rabin test that is deterministic, not probabilistic. This version uses a set of (small) primes as bases a, instead of choosing them randomly. For small primes pre-calculated prime tables can be used e.g. via trail division or a sieve method. However this deterministic version assumes that the generalized Riemann Hypothesis holds. This is an unsolved question. The larger the prime candidate the larger the set of prime bases a. This project

focuses on the *probabilistic* Miller-Rabin test.
A sieve procedure[6] e.g. sieve of Eratosthenes, identifies from a set $i, i+1, i+2, ...i+J$ those integers that are divisible by primes up to a selected limit. The divisible integers are removed and the indivisible primes remain. Sieves are useful if all primes within a range should be found. This is typical to create prime lookup tables. Cryptographic algorithms typically need only one or very few primes so sieve procedures that generate a large amount of primes do not make sense for this project.
The Miller-Rabin test is often used because it is simple to implement and a relatively fast algorithm. The quality of the resulting (probably) primes is sufficient for cryptographic purposes and can be adjusted with the number of iterations of the test.

2.1 Pseudo Code Miller-Rabin Test

This generic algorithm description (see Algorithm 1) is taken from an algorithms book [8].

Input 1: (approved) Random Bit Generator
Input 2: n, $n > 3$ an integer to be tested for primality; This can be e.g. p,q or an auxiliary prime p_1, p_2, q_1 or q_2 for DSA
Input 3: iterations k, the number of repetitions for test that determines the accuracy of the test.

Output: Either PROBABLY PRIME or COMPOSITE

```
rewrite n − 1 as 2^s · d where d is odd by factoring powers of 2 starting
 with n − 1
LOOP: for j = 1 ... k do
    generate random base a in the range [2, n2] using the RGB
    x ← a^d mod n
    if x = 1 or x = n − 1 then
        do next LOOP
    end
    for r = 1 ... s − 1 do
        x ← x^2 mod n
        if x = 1 then
            return COMPOSITE
        end
        if x = n1 then
            do next LOOP
        end
    end
    return COMPOSITE
end
return PROBABLY PRIME
```

Algorithm 1: Pseudo Code Miller-Rabin test

The Java implementation of this central part can be found in implementation section, see Listing 3.1.

2.2 Accuracy Miller-Rabin Test

The Miller-Rabin test [3] repeats itself with multiple random bases a for k iterations. With each round the probability that the candidate is indeed prime is increased. This can be improved indefinitely until the desired confidence level is reached. FIPS-186-4 [6] defines minimum requirements on the number of iterations for DSA, see Table 2.1.

Further it suggests to combine the Miller-Rabin test with one Lucas test

Table 2.1: Minimum of Miller-Rabin test iterations for DSA [6]

DSA Parameters	Miller-Rabin test iterations	Error probability
p:1024 q:160 Bits	p and q: 40	2^{-80}
p:2048 q:224 Bits	p and q: 56	2^{-112}
p:2048 q:256 Bits	p and q: 56	2^{-112}
p:3072 q:256 Bits	p and q: 56	2^{-128}

(another probabilistic test method). If the Miller-Rabin test is followed by one Lucas test the number of iteration for the Miller-Rabin test can be reduced to less than half of the iterations with the same error probability. The Lucas test

is out of scope of this project.
The PrimeApp uses 64 rounds of the Miller-Rabin test for *all* Bit lengths, this results in an error probability of $2^{-}128$ or less. This low risk level of generating a false positive (a candidate that is not prime, but is recognized as probable prime) is accepted for this project.

Chapter 3

Implementation

The implementation consists of the User Interface, the candidate number generation, the Miller-Rabin prime test provided by the Spongy Castle library (details later in section 3.5) and the newly implemented Miller-Rabin prime test. The created PrimeApp has about 350 lines of code, but takes significant time to run for large primes.

3.1 Overview and Process Description

The Project is implemented in a single Java class `MainActivity`. It contains several methods:

1. *GenerateProbablePrime*(↓`Bit length`, ↑`Output string`)
   Generates and tests one probable prime, finally performance timings are written to a Comma Separated Values (CSV) file.

2. *getRandomPrimeCandidate*(↓`Bit length`, ↓`Random Bit Generator`, ↑`Candidate prime`)
   Generates a candidate prime of the specified Bit length and returns it to the caller.

3. *MyMRPrimeTest*(↓`Candidate prime`, ↓`Number of test iterations`, ↓`Random Bit Generator`, ↑`is-probable-prime`)
   Executes a Miller-Rabin test on the prime candidate for the specified number of times (64). The result is a binary decision if the candidate is probably prime or not prime.

4. *onCreate*(`...`) Is executed when the PrimeApp starts and contains initialization code and event handling for user interactions.

Figure 3.2 illustrates the generation of a prime candidate. As Random Bit Generator `java.security.SecureRandom` was selected because the documentation of the Java standard library states that is self-seeded and generates True

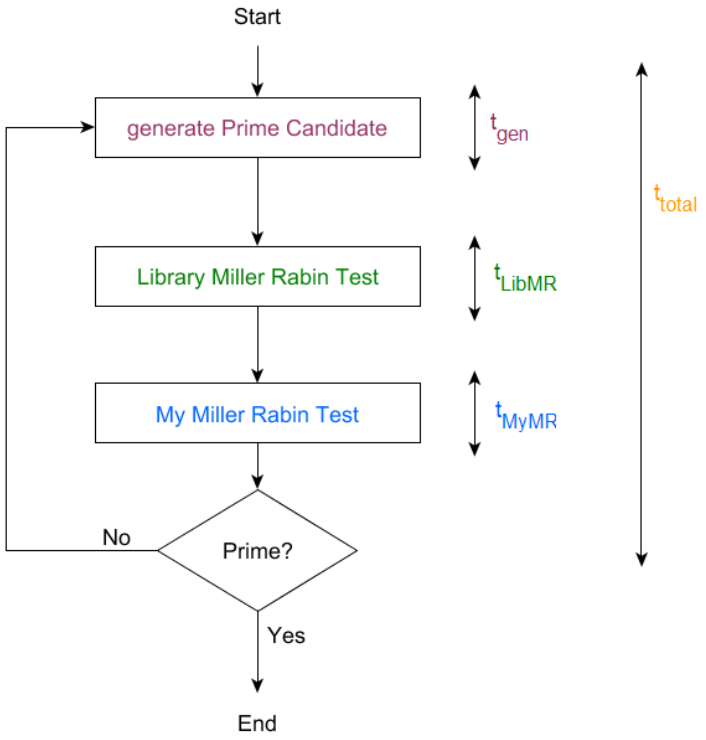


Figure 3.1: Process of generating a single prime with timing indications

Random Numbers, or more precisely True Random Bit sequence. This random number has m Bits is then hashed with a one way hash function. The number m depends on the hash function. In this implementation it can be 256 or larger, 256 was chosen for optimal distribution results of the hash function. The prime generation standard FIPS 186-4 [6] requires an NIST approved hash function listed in a special publication of NIST [1]. The well known algorithm *SHA256* is used in this project. It is considered as a strong hash function. The purpose of using the hash value of the random number is to ensure that the process can not be reverted. It could be potentially harmful if an attacker could influence the prime generation in a way so that the attacker would know or be able to manipulate the finally generated prime that is later used for encryption, decryption, key negotiating or a digital signature. Of course this depends strongly on the cryptographic algorithm that uses the prime.
The output of the SHA256 hash function is always 256 Bits long. The prime candidate can be 4, 8, 1024, 2048 or 3072 Bits long. For the larger primes the hashing has to be done multiple times in order to get enough bit material. The outputs of the hash function are concatenated to the desired length. For 4 and 8

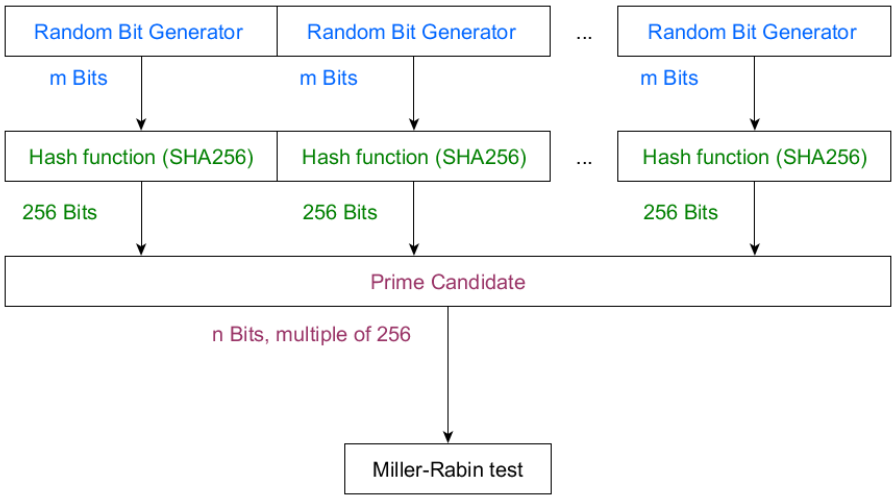


Figure 3.2: Process of prime candidate generation

Bit primes the 256 Bit output of one hash function is enough, in fact it has to be trimed to the desired smaller Bit length. The generation additionally sets both the Most Significant Bit (MSB) and the LSB. The generated candidate could have an MSB equal to zero then the candidate would not need the complete Bit length. An arbitrary number of zeros could be prepended (left) to the candidate without changing the value. The LSB is overwritten with 1 to reassure that the candidate will be odd. Then one prime candidate is generated and will be passed on to the Miller-Rabin test to check if the candidate is (probably) prime or not.

3.2 User Interface

The User Interface (UI) for the PrimeApp was designed and developed with Android Studio, see Figure 3.3. Many predefined UI element exist. This simple UI provides the essential elements for input and output.
In general for Android Apps the UI is defined for each `Activity` (a screen) independently in an separated xml file, activity_main.xml in this case. The PrimeApp consists only of one `Activity` so only one file for defining the UI is used. The elements like buttons including position, initial values and an ID are defined there. Then the Java code accesses the UI elements by searching for the element's ID and returns e.g. a button object that is used for event handling or setting output texts.
Figure 3.3 shows the start screen (`MainActivity`) of the PrimeApp and its UI elements. A information text is displayed and is not changeable in a `TextView`.

The `NumberPicker` lets the user choose from a predefined list of string values of prime Bit lengths. The picker then returns the value of the user's choice which updates a global variable in an value-change-event. During an calculation of a prime the Bit length value can not be changed. Additionally there is a text field `TextView` for displaying the output, e.g. generated prime number and the time used for the generation. The user can not change the values of these text fields. There are two buttons. One generates a single prime with the chosen Bit length and sets the output string to be displayed in the output text field. The outputs are

1. the prime number,
2. the Bit length N of the generated prime,
3. the number of attempts (`i`) that was needed until a prime number was found,
4. the result of the Spongy Castle Library's prime test "LibMR"[1], see section 3.5, (the test can be passed or failed),
5. the result of the created prime test "MyMR" (passed or failed),
6. the time of prime generation and "MyMR" test,
7. the time of prime generation on "LibMR" test in milli seconds.

The second button "Generate 1000 Primes" uses the same calculation routine, but generates 1000 primes numbers instead of one. Since one large prime already fills a screen of the mobile phone even with a small font, the 1000 primes are not displayed due to screen space limitations. However, the timing data is written to a CSV file for statistical processing outside of the app.

When this button is pressed a progress bar appears to inform the user about the status of the time intensive calculation. The current progress is displayed in per cent and also in absolute values, how many of the 1000 primes were already generated. This is done with the `ProgessDialog` class. It requires that the calculation is outsourced in a separate thread. This thread then does one calculation and updates the status of the progress bar in the UI-thread. UI objects can not be accessed directly from the calculation thread (or any other). However, there is a predefined method `runOnUiThread()` that is used to print out errors if any would occur and also to print out the result string. The result string contains the CSV file path and the total time for all 1000 primes including both Miller-Rabin tests.

[1]The library is forced to run 64 iterations of the test just like the new implementation "MyMR", hence the comparison is fair.

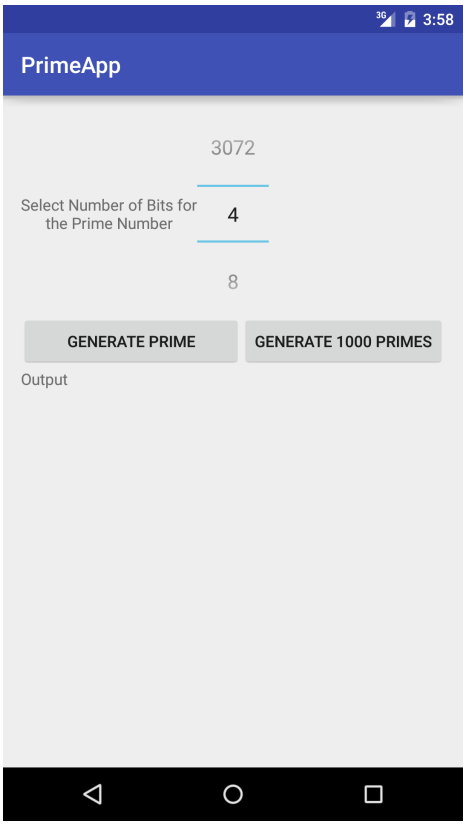


Figure 3.3: Start screen of the PrimeApp

3.3 Implementation of Miller Rabin Test

The "My Miller-Rabin" test starts in this implementation with defining the constants 0, 1 and 2. This is not trivial, because primitive data types in Java e.g. `integer` (32 Bit signed two's compliment) or `long` (64 Bit signed two's compliment) are far too small to store a large prime with up to several thousand Bit. This application needs a data type that can store arbitrary long integers, called *BigInteger*. This does not have to be implemented on Android because it is available in the mathematic package of Java (`java.math.BigInteger`). *BigInteger*s could be implemented as dynamically growing list of digits.
Then two plausibility checks follow, first if the candidate is smaller than two it is disqualified. Second, the least significant Bit is checked, if it is zero then the candidate is an even number is therefore divisible by two and can per definition never be prime (two is neglected for cryptography). Even numbers are early recognized as not prime. This does not require a expensive modulo operation, but only a one Bit check.
The *BigInteger* math package offers several handy operations e.g. `modpow()`

that is optimized for potentiating a *BigInteger* and performs a modulo operation in one step. This is a very important optimized operation that is used twice in the algorithm. Furthermore simpler operations also have to be used from the BigInteger package.
For comparisons with a constant, the constant has to be created first and then compared with the `equals()` method. Subtractions or additions also need to call the corresponding methods instead of the operators for primitives of the programming language. Only the loop indexes can still be handled with standard operators.

The implementation of the central Miller-Rabin test is shown in Listing 3.1. It adds a check if the candidate is larger than two to avoid testing negative integers and it tests if the candidate is odd by checking the Least Significant Bit (LSB).
Notice that in Listing 3.1 the candidate n - 1 is factored into its powers of 2 with a while loop, compare the pseudo code in section 2.1. After that the two boxed for-loops follow that do the essential Miller-Rabin test. The outer for-loop is the "iteration" loop. It repeats the test for several iterations (for this project fixed to 64 for both implementations). Then the random base is generated. The inner for-loop is responsible for repeatedly calculating x for the number of Bits the candidate consists of. If x is equal to 1 or equal to n-1, n is discovered as composite, otherwise the loop is repeated. If after all repletions of the test n is still not recognized as composite the result probable prime is returned.

Listing 3.1: Java source code excerpt of new Miller-Rabin test implementation (MainAcivity.java)

```
1     /**
2      * My Miller Rabin Test
3      *
4      * @param n           prime candidate
5      * @param random      Random bit generator
6      * @param iterations quality parameter that determines
           how likely prime the candidate is
7      * @return true if n is probable prime, otherwise flase
8      */
9 boolean MyMRPrimeTest(BigInteger n, SecureRandom random, int
      iterations) {
10
11         final BigInteger ZERO = BigInteger.ZERO;
12         final BigInteger ONE = BigInteger.ONE;
13         final BigInteger TWO = new BigInteger("2");
14         if (n.compareTo(TWO) == -1) { // n < 2 are forbidden
15                 return false;
16         }
17         if (n.testBit(0) == false) { // if LSB == 0 then n
               is a even number -> forbidden
```

```
18                  return false; // trivialy divisible by 2
19          }
20          int s = 0;
21          BigInteger d = n.subtract(ONE);
22          while (d.mod(TWO).equals(ZERO)) {
23                  s++;
24                  d = d.divide(TWO);
25          }
26
27          for (int i = 0; i < iterations; i++) {
28                  BigInteger a = BigIntegers.
                        createRandomInRange(TWO, n.subtract(TWO),
                         random); // random a in range [2,n-2]
29                  BigInteger x = a.modPow(d, n);
30                  // x = a^(d) mod n
31                  if (x.equals(ONE) || x.equals(n.subtract(ONE
                        ))) {
32                          continue;
33                          // no proof for composite found
34                  }
35                  int j;
36                  for (j = 1; j < s; j++) {
37                          x = x.modPow(TWO, n);
38                          // x = a^(2) mod n
39                          if (x.equals(ONE)) { // x == 1
40                                  return false;
41                          }
42                          if (x.equals(n.subtract(ONE))) {
43                          // x == n-1
44                                  break;
45                          }
46                  }
47                  if (j == s) {
48                          return false;
49                          // x == n-1 did not occur ->
                                composite
50                  }
51          }
52          return true; // probably prime
53  }
```

In C the compiler can be instructed to run different optimization levels. Java does not allow this. The optimizations in Java are mostly done by the Just In Time compiler at runtime. It will then know the environment and decide the optimizations depending on the system architecture.

3.4 Performance measurement

When the PrimeApp is started, it checks if the `PrimeTimeMeasurements.txt` exists. If yes, it is deleted and created new, otherwise just created. As soon as the user clicks one of the buttons a `FileWriter` class is created for writing the measured timings to the CSV file. The generation routine appends the measured times to the CSV one by one for each prime. Finally when the generation is complete a newline is appended and the data is persisted (flushed). After that the file writer is closed. If the user clicks another button a new file writer is created just as described.
As depicted in Figure 3.1 there are three essential parts: the prime candidate generation, the Miller-Rabin test implemented by the library and the newly implemented "My Miller-Rabin" test. All three sum up to the total time of one prime generation, which can be different from the total time displayed to the user. This user displayed time is calculated end time of the generation minus start time of the generation. Particularly if the smart phone goes into a sleep state and the app is suspended, but then this time still increases. Further more memory allocation times or file access times are included, yet in the CSV file the plain calculation time stored. This also described in section 3.2.
For the performance measurement the times shown in Figure 3.1 are used. The durations t_{gen}, t_{LibMr} and t_{MyMR} are calculated in each attempt to find a valid prime. Until one prime is found it is likely that several attempts have to be made (typically up to a few hundred). The number of attempts has a major influence of the performance of the generation. In this project the two different implementation of the Miller-Rabin test are used with the *same input*, the prime candidate, to ensure equal circumstances for the tests. Hence the time each version needs, includes the generation time and the according testing time:

$$t_{MyMRtime} = t_{total} - \sum_{i=1}^{attempts} t_{LibMR_i}$$

and respectively

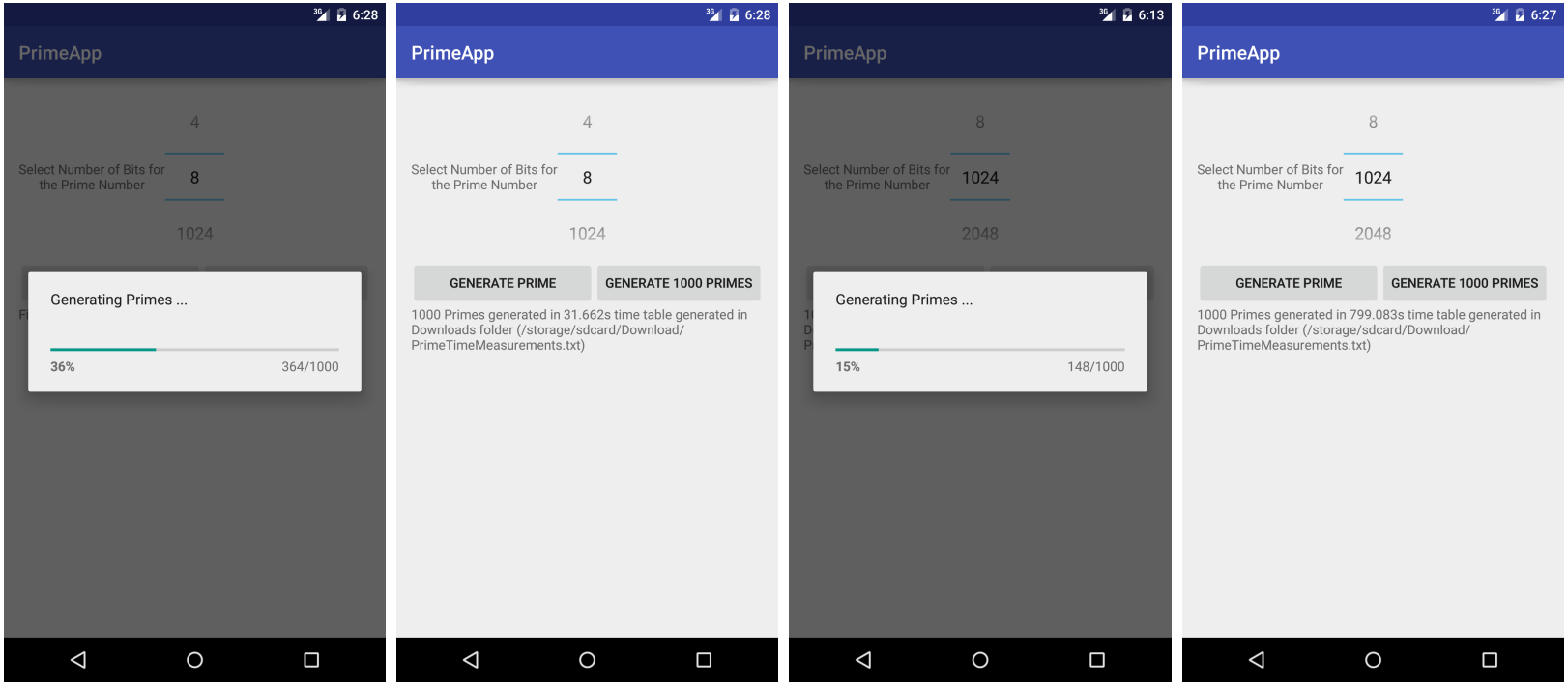
$$t_{LibMRtime} = t_{total} - \sum_{i=1}^{attempts} t_{MyMR_i}$$

So the time of each version is *freed from the time the other test takes.* $t_{MyMRtime}$ and $t_{LibMRtime}$ are the times that are needed for (repeatedly) generating and checking prime candidates until one probable prime is found. These times are stored in the CSV files. Hence a generation of 1000 primes, results in 2000 entries in the CSV file in the format:

$< t_{MyMRtime}$ of prime 1>, $< t_{LibMRtime}$ of prime 1>; ...

$< t_{MyMRtime}$ of prime i>, $< t_{LibMRtime}$ of prime i>;

where i has the maximum of 1000. The statistical processing of the measured times was then done with an external program that takes the CSV files as input.



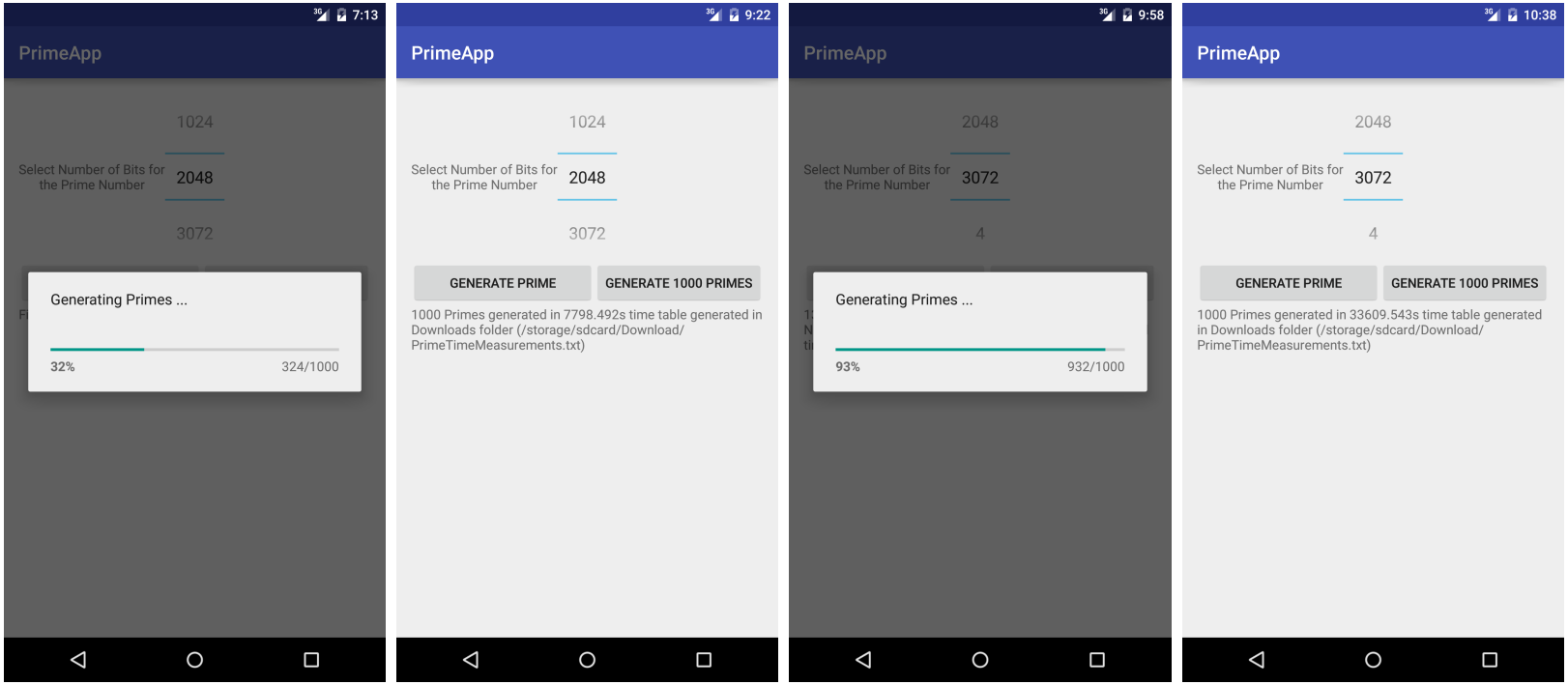
(a) Loading-Screen for generating 1000 8 Bit primes

(b) Result-Screen for generating 1000 8 Bit primes

(c) Loading-Screen for generating 1000 1024 Bit primes

(d) Result-Screen for generating 1000 1024 Bit primes

Figure 3.4: Screen shot series of PrimeApp - Android *Emulator*



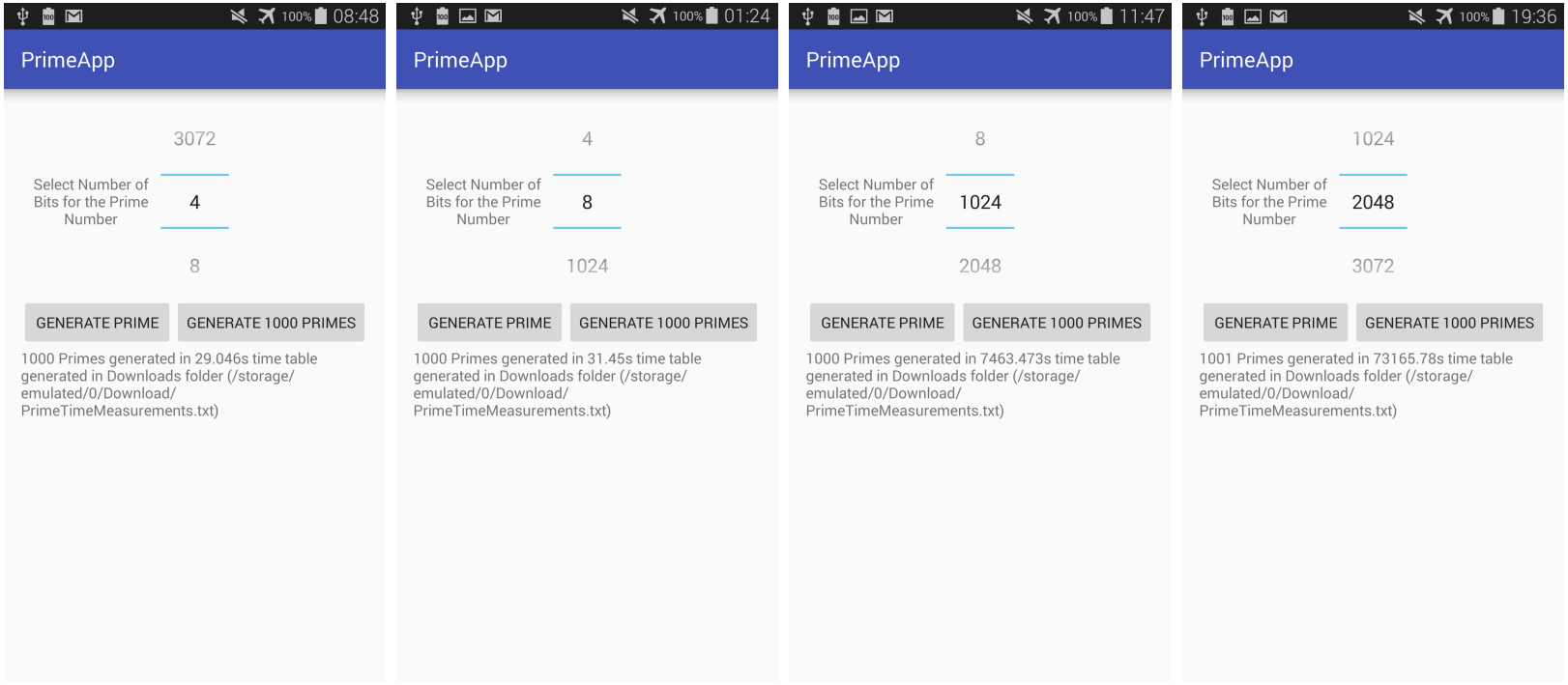
(a) Loading-Screen for generating 1000 2048 Bit primes

(b) Result-Screen for generating 1000 2048 Bit primes

(c) Loading-Screen for generating 1000 3072 Bit primes

(d) Result-Screen for generating 1000 3072 Bit primes

Figure 3.5: Screen shot series of PrimeApp - Android *Emulator*



(a) Result-Screen for generating one 4 Bit prime

(b) Result-Screen for generating one 8 Bit prime

(c) Result-Screen for generating one 1024 Bit prime

(d) Result-Screen for generating one 2048 Bit prime

Figure 3.6: Screen shot series of PrimeApp - *Samsung Galaxy S5*

3.5 Cryptography Library: Bouncy Castle

The Bouncy Castle [4] cryptography library is a open source and non-profit project since 2000. It is available for Java and C# under MIT License. The project originates in Australia and is therefore *not* limited by artificial restrictions on key length or encryption strength. American restrictions on the export of cryptographic software do not apply for this library, nor by any other country (Australia does not yet impose restrictions on cryptography). The Bouncy Castle Library contains a variety of cryptographic relevant algorithms e.g. hash functions, message authentication codes, public key cryptography standards, block- and stream-ciphers, key generation and exchange implementations. The prime generation component is used to validate the newly created prime generation implementation.
The Bouncy Castle Library is partly included in the Android Operating System. Therefore the Bouncy Castle Library can not be included completely to an android app as-is due to class name conflicts. In 2014 the library was re-released by a third party under the name Spongy Castle to avoid this naming problem. The *PrimeApp* uses the method `org.spongycastle.math.Primes.isMRProbablePrime(...)` [2] to test the *PrimeApp* prime test method `MyMRPrimeTest(...)` to avoid implementation bugs, both use 64 iterations for all tests in this project. However, the number of iterations is parameter to the method and is decided by the crypto algorithm that uses the test.
In order to be able to use the Spongy Castle Library it has to be enrolled in the System's Security manager (Android) by inserting the library's Security Provider. The library was downloaded from Github [7] as Java Archive (.jar) and then copied into Android PrimeApp project's libs folder. Note that the mere copy is not enough, also the builder tool (gradle) needs the .jar file(s) mentioned in a configuration file in order to compile the library into the project. Particularly the lines:

- compile files('libs/core-1.54.0.0.jar')
- compile files('libs/pg-1.54.0.0.jar')
- compile files('libs/pkix-1.54.0.0.jar')
- compile files('libs/prov-1.54.0.0.jar')

have to be added under the dependencies block in the `build.gradle` build configuration file.

Chapter 4

Manual

This chapter describes how to set up and install the PrimeApp and also gives a user guide on operating the app. The installation file is a .apk (app-debug.apk) file which the user needs to copy on the device and simply click it to install it. When the app is started the user selects the desired bit length of the prime to be generated and either generates one or 1000 primes with on of the two buttons.

4.1 Installation

The product of this project in an Android app in the form an .apk file. In order to install the app the user has to copy the .apk file on an Android device e.g. smart phone, tablet or an emulator of such a device. This copy is easiest via an USB cable, but if the device is connected to the internet it could also be sent via the wireless network e.g. as Email attachment.
If a physical device is used, apps that are not downloaded from the Google Play store are by default forbidden for security reasons. So third party applications, like PrimeApp, have to be allowed before they can be installed under *Settings - Security - Unknown sources* with Android 4.0+. This option must be tabbed to enable (allow).
Then the user should navigate to the .apk file on the device and tab it. An installation menu will appear. The user should then confirm the installation by tabing OK/Continue. After the installation the PrimeApp can be started directly by tabing "open". Additionally the app is listed in the systems app list with a green Android symbol and the name "PrimeApp". Tabing this icon also starts the app. If an emulator is used, third party apps might be enabled by default. There are many Android emulators available, one developer friendly made available by Google[1] was used for testing, another more user friendly emulator is Nox[2] with the Cyanogen Mod as Firmware (ROM). The emulator has to be downloaded and installed on a PC or Laptop, then it has to be started. The

[1] https://developer.android.com/studio/run/emulator.html
[2] http://en.bignox.com/blog

same procedure to install PrimeApp has to be applied. For the Nox emulator the user has to drag and drop the .apk file in a folder e.g. start the file manager and select the downloads folder. When droping the file into the folder the emulator asks the user to activate root access. This has to be accepted. The PrimeApp itself does not need root permissions, but it is needed for the copy process. The emulator restarts and then again the .apk file should be dropped into the Downloads folder or home screen. The emulator then lets the user choose to store the app into the emulator's app folder e.g. `C:\Users\myusername\Nox_share\App`. If the user then double clicks "open app folder" the .apk file opens in the Android file browser. From here on the steps are the same as on the physical device. Tab the .apk file and confirm the installation.
The PrimeApp needs read and write permissions for the SD-card in order to store the measured timing data.

4.2 User Guide

After the installation is completed, the user can start the app, either they open it from the installation screen with the open button or they select the PrimeApp from the app list. Then the home screen appears as shown in Figure 3.3. The UI is designed to be self explanatory. First the user has to select the desired bit length of the prime(s) to be generated, by default 4 is selected. Primes with 4 Bit are meant as plausibility test. There are only two primes that need exactly 4 Bits to be stored, 11 and 13. Then the users can choose if they want to generate only one prime by selecting the left button labeled "Generate Prime" or 1000 primes by tabbing the right button labeled "Generate 1000 Primes". Depending on the selected Bit length the generation of one single prime can take up to one minute (!). The app may seems to have crashed, but typically did not crash. All exceptional errors will result in an error message in the output field. If an error message appears, try again or restart the application. However, this is unlikely. If the users decide to generate 1000 primes with the selected Bit length the generation can take much longer than only one minute. Therefore a loading screen appears, see Figure 3.4, picture (c). On the left bottom the current progress is shown as percentage. In the right bottom the exact number of generated primes and the maximum (1000) are displayed.
When the generation is completed, the loading screen disappears and the the output field displays the result of the generation. For generating 1000 primes the path of the timings CSV file is shown and also the total time from pressing the generate button until the result is determined appears[3]. Notice in the screen shots that the path varies from device to device.
If only one prime is generated the time that the library implementation took and the time that the new implementation needed are shown. Additionally the

[3]Note this is different from the time t_{total}. This user displayed time includes all tasks the app does. It particularly includes the time needed to write the CSV file which is typically a "slow" operation in comparison to CPU calculations. If the device goes into sleep mode, the sleep time is also included.

Bit length and the number of candidates needed until a prime was found are displayed. Just to be sure the primarily test was indeed passed by both the library Miller-Rabin test as well as the new implementation of the Miller-Rabin test, the results are displayed. In both cases for single prime or multiple prime generation the timings file is written. The single prime generation does not have a loading screen, the generation for multiple primes does.
The app can be stopped by simply tabbing the back button of the virtual or real device.

Chapter 5

Experimental Results

In this section the duration that is needed to create 1000 primes of the respected bit lengths are shown and analyzed.
The new implementation MyMR was tested on both an emulator and a physical device. The development environment Android Studio comes along with an Android Emulator, in this case version 25.1.7. As virtual device a Nexus 5 with Android 5.0 (Lollipop, API level 21) was used and executed on a standard workstation Notebook under Windows 10. The physical device is a Samsung Galaxy S5 with Android 4.4.2 (KitKat API 19).
Plotting the time needed to generate each of the primes can be found in Figure 5.1, exemplary for 1024 Bit primes. This Diagram shows that there are few outliers that take an extraordinary large amount of time, while others take very little time. The representation in Figure 5.1 shows that roughly, but there are better statistical representations of the results.
All the primes timing statistics are visualized with a Box-plot (or Whiskas-plot). Each plot was generated with the timings of generating 1000 primes. The Box-plot shows (from bottom to top) the minimum, the first quantile, the median, the third quantile and the maximum. The first quantile allows the statement 25% of the data values are below the first quantile limit. The second quantile (or median) allows the statement 50% of the data values are below the median. Similarly the third quantile allows the statement 75% of the data values are below the third quantile limit.
The 4 and 8 Bit primes are intended to plausibly show that valid primes are generated. They are small primes that are not usable for encryption and have therefore a low calculation cost (time). Figure 5.2 shows the Box-plots of the 8 Bit primes. 4 and 8 Bit primes are not further analyzed.

However, 1024 Bit large primes are definitely relevant for encryption purposes. Figure 5.3 reveals that the Android emulator is significantly faster with generating 1024 Bit primes than the physical device. This is a surprising result, since one might expect the situation to be reversed. The exact statistical values for the Box-plots can be read in Table 5.3 and Table 5.4. All values from the

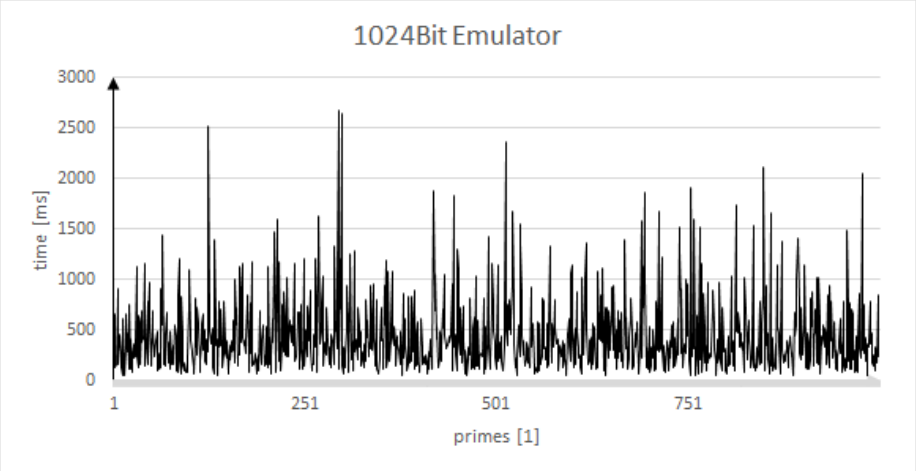


Figure 5.1: Graphical representation of the timing data of the generation of 1000 primes with 1024 Bit length

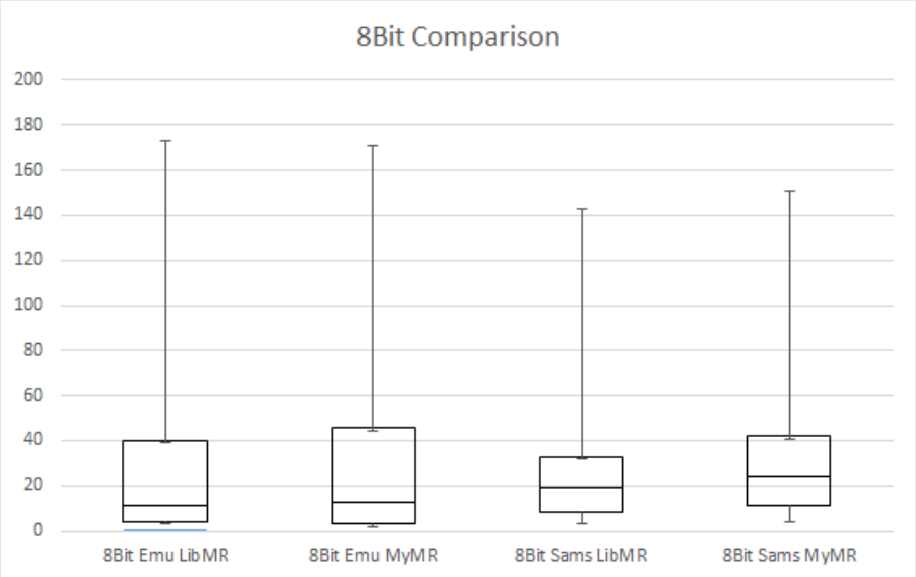


Figure 5.2: Comparison of 8 Bit prime timings using the library and the MyMR implementation on an emulator and a physical device.

emulator are lower than the corresponding values from the real device(!). The generation with the library test takes a (median) time of 331.5ms on the emulator and 2.82s on the physical device, which is 8.5 times longer. For all four cases the maximum (longest calculation) is very large in relation to the rest of the calculation durations. This means there are some significant outliers.
In order to better compare the the MyMR implementation with library implementation, Figure 5.4 shows only the two (faster) results from the emulator. Visually there is almost no difference, except the minimum is slightly lower for the library implementation. The exact values from Table 5.3 reveal that the library implementation is in all parameters slightly faster than the MyMR implementation expect with the minimum MyMR is better.

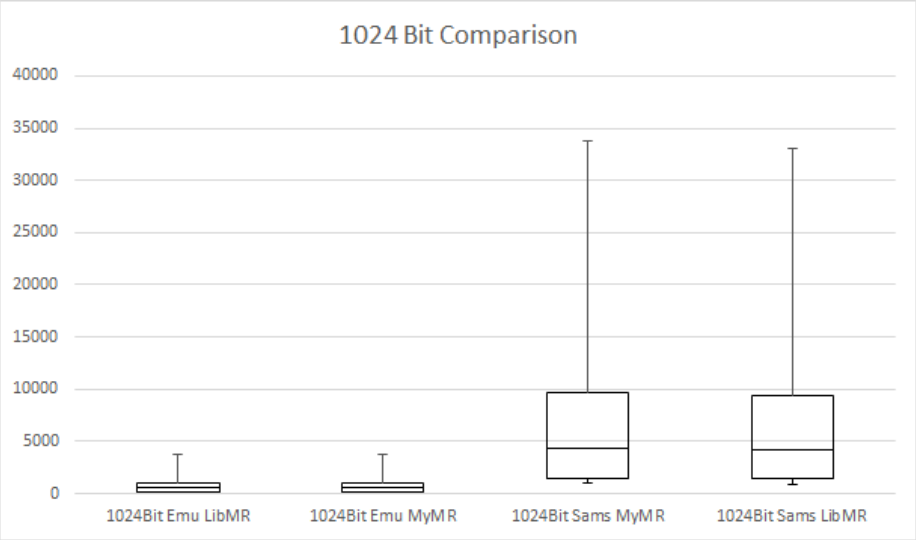


Figure 5.3: Comparison of 1024 Bit prime timings using the library and the MyMR implementation on an emulator and a physical device.

The next larger Bit length for primes is 2048. This is double the length, but not double the calculation time. For the emulator the average time of one prime generation is roughly the tenfold for the doubled bit length: 1024 Bit takes 430ms (libMR) and 441ms (MyMR) where 2048 Bit primes need on average 3993ms (libMR) and 4027ms (MyMR). This indicates a non-linear nature, which was expected. Figure 5.5 depicts the generation of the four data sets for 2048 Bit of both the library and MyMR implementation on the emulator and the real device with similar drastic differences of the emulator and the physical device.

Figure 5.7 shows comparison the library performance and MyMR performance on the emulator, again they are very similar. The optical impression of both data sets is the same. Only when the exact numbers are compared in Table 5.4 minimal differences can be spotted. There are almost no differences between the library- and the MyMR-performance, but again the real device is

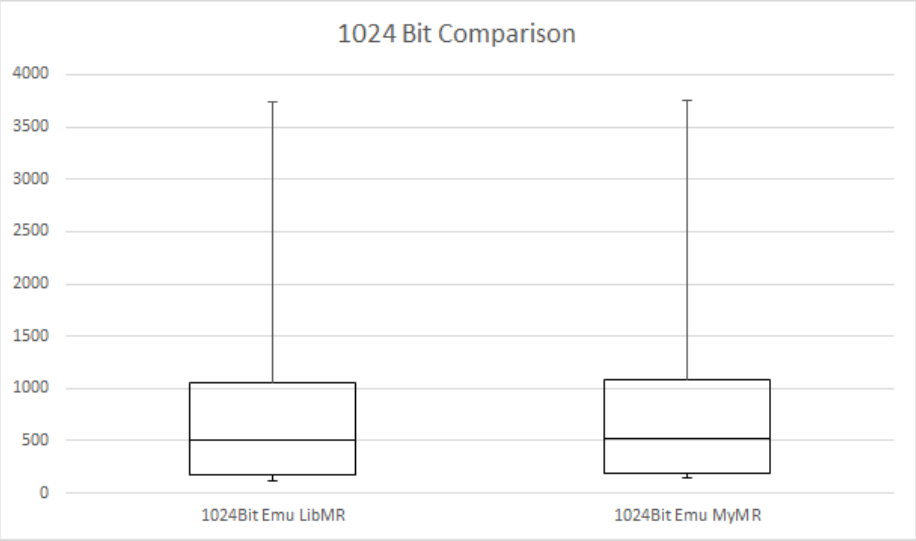


Figure 5.4: Comparison of 1024 Bit prime generation timings for 1000 primes using once a library primarity-test and once MyMR

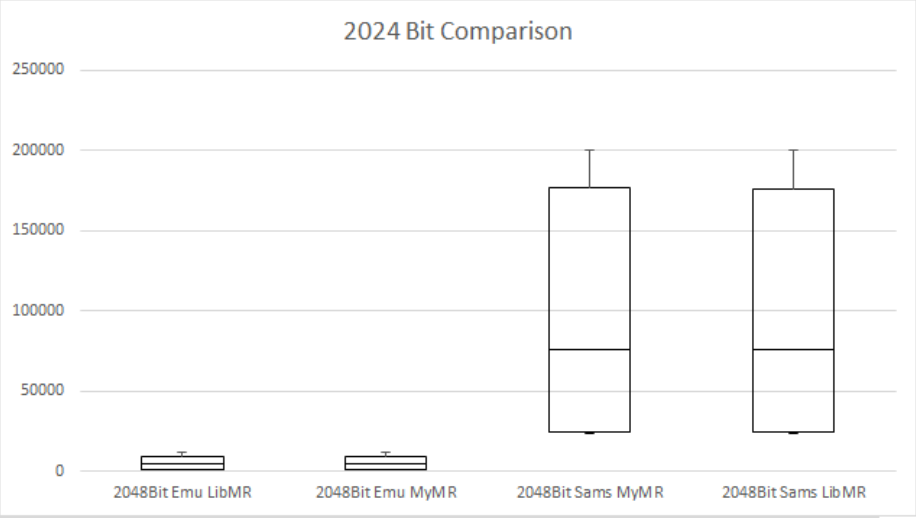


Figure 5.5: Comparison of 2048 Bit prime timings using the library and the MyMR implementation on an emulator and a physical device.

Figure 5.6: Comparison of 2048 Bit prime generation timings for 1000 primes using once a library primarity-test and once MyMR

Table 5.1: My caption

Bit length	Emulator calculation time	Samsung Galaxy S5 calculation time
8	32s	31s
1024	799s (=14min)	7463s (=2h)
2048	7798s (=2h)	73166s (=20h)
3072	33610s (=9h)	>4 days

slightly faster, by 0.36%, compare Table 5.4 row "Average", columns "3072 Bit Emu LiBMR" and "3072 Bit Emu MyMR". Unfortunately there are no values for 3072 Bit prime on the physical devices because the time needed for execution exceeded 4 days. This behavior was already expected from the results of generating 1024 and 2048 Bit primes on the device, where the execution on the samsung device took much longer than on the emulator.

Table 5.1 shows the user displayed times for the generation of 1000 primes including all tasks of the app, summarized from the screen shot series. There is no exact data for 3072 Bit on the real device due to the long calculation time.

Table 5.2 compares the time needed to generate prime candidates vs. the time needed to execute the Miller-Rabin test. For getting a rough overview of the time distribution between the two tasks five data points were collected for 1024 and 2048 Bit primes. The result is clearly that the Miller-Rabin test takes the majority of the time about 85% and the generation takes only about 15% for 1024 Bit primes. For 2048 Bit primes the ratio shifts towards the testing. With the doubled Bit length the testing time clearly dominates with about 95%

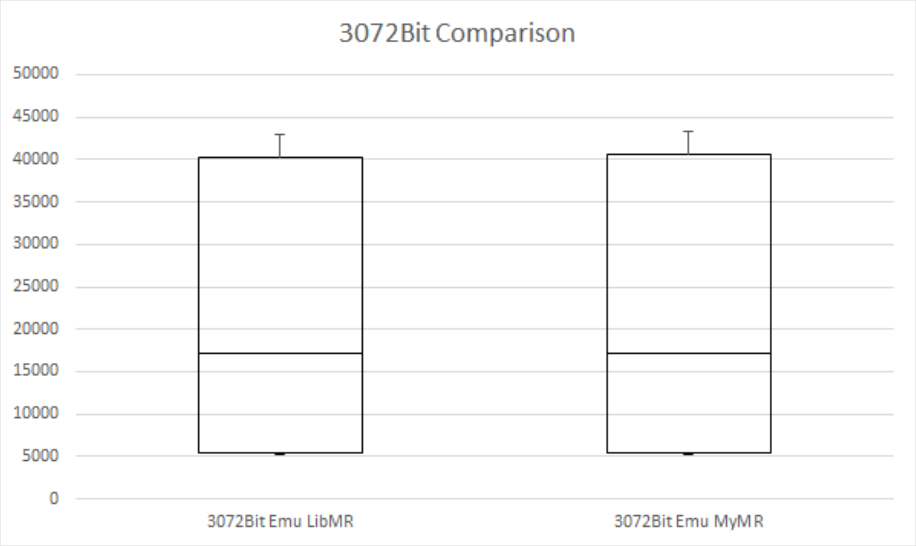


Figure 5.7: Comparison of 3072 Bit prime generation timings for 1000 primes using once a library primarity-test and once MyMR

of the total time. This indicates that for 3072 Bit primes the ratio will further be dominated by the testing task. Moreover, the generation of double the Bit length of candidates only tripled, while the testing time of the doubled Bit length is about the ten-fold. The statistics of Table 5.2 might have a only a low accuracy due to the low number of calculations and the previously observation of strong outliers. However, it is safe to say that the testing takes the majority of the total time.

Table 5.2: Calculation time comparison on the emulator: t_{gen} vs. t_{MyMR}

	calc. 1 [ms]	calc. 2 [ms]	calc. 3 [ms]	calc. 4 [ms]	calc. 5 [ms]	Average of 5 calc [ms]	Time distri-bution [%]
1024 Bit: t_{gen}	93	623	968	345	347	475,2	14,77
1024 Bit: t_{MyMR}	690	3557	5354	1833	2272	2741,2	**85,23**
2048 Bit: t_{gen}	907	1222	575	3873	440	1403,4	5,30
2048 Bit: t_{MyMR}	16365	21140	10614	68605	8635	25071,8	**94,70**

Table 5.3: Statistical parameters of the recorded timing data for 4 Bit, 8 Bit and 1024 Bit primes

	4Bit Emu LibMR	8Bit Emu LibMR	8Bit Emu MyMR	8Bit Sams LibMR	8Bit Sams MyMR	1024Bit Emu LibMR	1024Bit Emu MyMR	1024Bit Sams MyMR	1024Bit Sams LibMR
1.Quatile	4	3	3	8	11	179	183.75	1448.5	1421.5
Median	14	7	9.5	11	13	331.5	338	2879.5	2822
3.Quartile	33	29	33	14	18	540.25	556	5284.75	5150
Max	192	133	125	110	109	2682	2683	24146	23618
Min	2	1	1	5	7	64	42	488	502
Average	23.04	18.29	20.52	13.37	17.20	429.95	440.75	3886.89	3808.88
Std. Deviation	19.03	16.95	18.08	5.73	7.61	256.61	267.18	2479.14	2420.60

Table 5.4: Statistical parameters of the recorded timing data for 2048 Bit and 3072 Bit primes

	2048Bit Emu LibMR	2048Bit Emu MyMR	2048Bit Sams MyMR	2048Bit Sams LibMR	3072Bit Emu LibMR	3072Bit Emu MyMR
1.Quatile	1367.5	1365.75	24398.25	24291.5	5341.75	5361.75
Median	2773.5	2813	51678.5	51550	11733.5	11797.5
3.Quartile	5152.75	5209.25	100539.5	100478.5	23188.25	23403.75
Max	32491	32761	630585	627882	128291	128535
Min	300	268	5993	5909	993	983
Average	3992.71	4027.44	73850.92	73653.63	16989.15	17049.69
Std. Deviation	2736.82	2766.33	51390.99	51232.31	12054.85	12097.57

Chapter 6

Summary

The product of this project is an Android app that generates primes. This project successfully implemented a prime number generation for large primes as they are approved by the NIST FIPS-186-4 standard [6] with lengths of 4, 8, 1024, 2048 and 3072 Bit lengths.
The small prime generation shows obvious primes to be generated correctly. Large primes are not only tested with this implementation but also with the Spongy Castle cryptography library implementation. So far all test runs completed without mistakes.
The method chosen in this project can be summarized as intelligently guessing a prime candidate and then afterwards testing it to show if it is really prime. This is repeated until a prime is found. The candidate is generated by generating random Bits and hashing (SHA256) them. This is repeated until enough Bits for the desired prime are generated. The hashed values are concatenated and the MSB and LSB are set. The MSB must be set so the candidate is in the correct number range and the LSB is set so that the candidate is odd, which is a trivial requirement for a large prime. The probabilistic Miller-Rabin test is applied for 64 iterations that results in an error probability of 2^{-128} or less which is conservatively chosen and approved by the FIPS-186-4 standard [6].
Moreover the performance of the prime generation was measured and evaluated in detail. 1000 primes of each Bit length were generated. Box-plots illustrate the statistics of the generation. Two major conclusions can be drawn from the statistical analysis: First, against the authors expectations the prime generation was significantly slower on the physical test device than on the emulator by factor between 8 and 18, for larger Bit length the factor will continue to grow. This is a surprising result that motivates further investigations. Second, the recreated implementation of the Miller-Rabin test performs almost as fast as the reference implementation (Crypto library). The reimplementation needed at most 2.45% longer than the library implementation, in three categories less than 0.87% longer. This is a satisfactory performance outcome. The result on the Android emulator is that on average a 1024 Bit prime takes 440ms, a 2048 Bit prime needs $\sim$4s and a 3072 Bit prime requires $\sim$17s to generate.

Chapter 7

Future Work

Primes could not only be generated and tested with arbitrarily high confidence, but can also be constructed with e.g. the Shaw-Taylor method that proves that the prime is indeed prime without doubt, in contrast to the probabilistic Miller-Rabin test. This construction method is more complicated, but ensures perfect quality of the prime. A performance evaluation and comparison to the results of the probabilistic method would certainly be beneficial.
The PrimeApp could introduce the loading screen also for single prime generation.
The performance evaluation showed that some few large outliers burden the average prime generation. There might be a potential speedup if that outliers could be skipped, however this may be hard to realize because there are typically already multiple different candidates tested. This would then essentially mean to improve the candidate selection with heuristics e.g. if the sum of the digits of n is divisible by three it is likely that n is also divisible by three and therefore early disqualified. This raises the question if that will perform better and if such heuristics would only disqualify composites or some primes too and the impact of that.
Performance of the generation is a key driver for cryptography. In order to achieve a speedup the Miller-Rabin test could be parallelized. The test implements n independent iterations of the test with random bases. Hence up to n workers could run the test in parallel. If all tests are done in parallel, the potential early recognition of composite numbers is lost, but the calculation time may be reduced (maybe significantly), assuming computing resources are available e.g. multi core CPUs or multiple CPUs, or both, which is the case for most modern Android devices. The trade off between speedup and processing power remains for mobile devices.
This project used 64 rounds of the Miller-Rabin test independent of the Bit length. Depending on the application less round may result in a sufficient accuracy. Also the rounds could be adjusted to the bit lengths, short bit length may require few rounds.
Additionally the 1000 primes generation could be parallelized to utilize the de-

vice optimally. However, this is rather academic since the user typically needs a low number of primes depending on the algorithms used e.g. DSA internally needs four primes, RSA two.
The strong performance differences between the emulator and the real device remain a mystery and motivate further investigations.

Chapter 8

Bibliography

[1] Elaine Barker. *NIST 800-57 Part 1 - Recommendation for Key Management*. 2016. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf.

[2] Peter Dettman. *Boucy Castle Prime Implementations*. 2015. URL: https://github.com/bcgit/bc-java/blob/adecd89d33edf278a5c601af2de696f0a6f65251/core/src/main/java/org/bouncycastle/math/Primes.java.

[3] Magorzata Engeleit. *Geeignete Algorithmen für elektronische Signaturen*. 2007. URL: http://www.iti.cs.tu-bs.de/~waetjen/Diplomarbeiten/Engeleit.pdf.

[4] Legion of the Bouncy Castle Inc. *Bouncy Castle Cryptography Libraray*. 2000. URL: https://www.bouncycastle.org/.

[5] Daniel Loebenberger and Michael Nsken. *NOTIONS FOR RSA INTEGERS*. 2012. URL: http://cosec.bit.uni-bonn.de/fileadmin/user_upload/publications/pubs/loenus11a.pdf.

[6] Patrick D. Gallagher NIST. *FIPS 186-4 Digital Signature Standard (DSS)*. 2013. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[7] Roberto Tyley. *Spongy Castle Cryptography Libraray*. 2014. URL: https://rtyley.github.io/spongycastle/.

[8] Wikibooks. *Algorithm Implementation - Mathematics - Primality Testing*. 2014. URL: https://en.wikibooks.org/wiki/Algorithm_Implementation/Mathematics/Primality_Testing.