Implementing an efficient randomized algorithm for testing equality of integers succinctly represented by arithmetic circuits with addition and multiplication.

Pierluigi Giulivi

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2021

Abstract

Comparing massive integers can be trivial if they are in binary, but when we represent them by arithmetic circuits or Straight-Line Programs (SLPs) with addition and multiplication as operations, deciding equality (EquSLP) becomes more challenging. The best-known algorithm presented by Schönhage 1979 is a polynomial-time randomised algorithm that lies in the complexity class co-*RP*, meaning that if the two SLPs we are comparing are not equal, there exists a probability for the algorithm to decide they are. The algorithm randomly chooses a large enough integer R and computes each arithmetic operation of both circuits modulo R. If both SLPs have the same result, we can conclude they are equal with a high probability; if not, we are sure they are not equal. We implemented and tested the algorithm and realised that its probability of failing decreases exceptionally quickly as we increase the value of R and can be bounded. We tested worse case scenarios and found that in many cases choosing R in the neighbourhood of 30 bits or more (i.e. $2^{30-1} \leq R \leq 2^{30}-1$) can yield very satisfying results, meaning we do not have to go well beyond even if the SLPs we are comparing are gigantic.

Acknowledgements

Throughout the research and writing of this dissertation, I got support from many people. Firstly, I would like to thank my supervisor Dr Kousha Etessami whose expertise guided me in formulating the research topic and methodology. I want to thank my parents and my sister for always supporting me. They are essential in my life and inspired me to follow a career in Computer Science. Finally, I want to thank my friends for bringing me joy and distracting me from my research.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Pierluigi Giulivi*)

Table of Contents

Chapter 1

Introduction

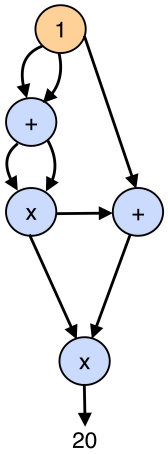
Numbers, symbols representing an arithmetic value, are used to count, perform calculations and solve problems. We use arithmetic operations to play with them and make new ones. First comes addition. Second we have multiplication, which is repeated addition ($9*3 = 9+9+9$). Third comes exponentiation, which is equivalent to repeated multiplication ($9^3 = 9*9*9$). Fourth? Fourth is stacked exponentiation 9^{9^9}, and like this, we can create an infinite set of operations that make our numbers grow bigger and bigger. It is easy to see that with few mathematical symbols, we can create colossal amounts. For example 9^{9^9} has 369693100 digits while the amount of "elementary particles in the observable universe" is around 85 digits (Aaronson 2021).

Computers are physically limited in their storage and processing power. Having immense numbers poses problems to modern machines when it is time to manipulate, compare or perform arithmetic operations. Modern computers have a structure of 64 bits, limiting their capability of playing with numbers that go well beyond this. Computers will use different approaches, such as 64-bit floating-point numbers or decimal representations, to do calculations with big numbers. These techniques become less efficient and prone to inaccuracy as the numbers grow.

A fundamental task while dealing with numbers is deciding if one is greater than the other or equal. This problem might sound trivial if both numbers are in binary, but getting the binary representation requires explicitly computing the value (hence finding the 369693100 digits), which is very inefficient. Instead, we will use a succinct way of representing integers called arithmetic circuits or straight-line programs (SLP).

1.1 Arithmetic Circuits

Let n be a positive integer, an arithmetic circuit or straight-line program (SLP) X with n inputs is a directed acyclic graph. X has n nodes with no incoming edges, the input nodes, and one output node with no outgoing edge. The other nodes are called gates and have an in-degree of two. An operator from the set of arithmetic operators labels the nodes that are gates (section 7.2.2 Arora and Barak 2009). We call the size of the arithmetic circuit the number of gates in the circuit; we do not count the input or output nodes. This project will focus only on arithmetic circuits with integers as inputs, and only addition and multiplication as arithmetic operations. For simplicity, we will have only one input node equal to integer 1. Starting with integer one and using addition and multiplication repeatedly, we can create any positive integer imaginable. Hence, our arithmetic circuits are simply integers broken down into many operations. We can see in figure 1.1 how we can represent the integer 20 as a directed acyclic graph or a sequence of operations.



$$x_0 = 1$$
$$x_1 = x_0 + x_0 = 1 + 1 = 2$$
$$x_2 = x_1 * x_1 = 2 * 2 = 4$$
$$x_3 = x_2 + x_0 = 4 + 1 = 5$$
$$x_4 = x_2 * x_3 = 4 * 5 = 20$$

Figure 1.1: Twenty as a directed acyclic graph and a sequence of operations.

1.2 Objective

We know that massive numbers pose problems to modern computers. This project aims to investigate one of these problems, namely comparing larger integers represented by arithmetic circuits. Arithmetic circuits allow for a more succinct representation of such large integers. Since we are representing integers differently, we need a different way of comparing them. Over many years of research, a number of techniques for comparing arithmetic circuits have been found but many of them have not been implemented and experimentally tested.

The objective of this project is to focus on arithmetic circuits with addition and multiplication exclusively. We want to be able to tell if two SLPs are equal or not. We will implement and test a functioning version of the algorithm described by Schönhage 1979 so that we can answer this question. This problem is called EquSLP (equality straight-line program). This algorithm is the most efficient known to date, but it is randomised, meaning that each time we run it, there exists a probability of getting the wrong result. This probability connects to a random component of the algorithm; understanding this random component and improving the probability of success is the main focus of research and experimental testing of the algorithm.

Research about these algorithms is of great importance as these problems connect to many Number Theoretic problems such as Polynomial Identity Testing (PIT). Making improvements or discoveries could have a significant impact in many other areas of active research. We will see many examples of this in the next chapter.

This project has five chapters, including this one, which is chapter 1 (The Introduction). In chapter 2 (Background), we provide a literature review and some core definitions. Chapter 3 (The Algorithm) introduces the algorithm together with an analysis of it and its implementation on Python. Chapter 4 (Experimental Research) tests the algorithm and gathers lots of data about the relation between the random component and the probability of failure, showing that relatively small random numbers can decrease the probabilities of failure significantly. Finally, chapter 5 (Conclusions), summarises the main findings of the study and provides recommendations for further research. Throughout this project, we will assume some mathematical maturity and basic understanding of computational complexity and algorithms.

Chapter 2

Background

In this chapter, we are going to introduce complexity classes. We will go over previous literature around the EquSLP problem, showing the profound significance of this problem and all the connections it has to many other open questions in Number Theory.

2.1 Complexity Classes

Throughout this project, we will use and discuss complexity classes. Therefore, we will briefly define and explain what we mean by this. Computational complexity theory is concerned with the amount of computing power necessary to complete a task. An algorithm's efficiency is measured by counting the number of basic operations needed to get a result as the size of the input grows. As Wigderson 2019 would defined it: "efficient computation ... determining the minimal amounts of natural resources (like time, memory, and communication) needed to solve natural computational tasks by natural computational models". All of this translates to the following mathematical definition:

Definition (The class Deterministic Time $DTIME$):
"Let $T : N \to N$ be some function. We let $DTIME(T(n))$ be the set of all Boolean (one bit output) functions that are computable in $c * T(n)$-time for some constant $c > 0$." (chapter 1.5 Arora and Barak 2009).

Definition (The class Polynomial Time P):
"$P = \bigcup_{c \geq 1} DTIME(n^c)$" where c is a constant (chapter 1.5 Arora and Barak 2009).

The complexity class P serves as an approximation to the class of decision problems that can be solved efficiently. These are some basic complexity classes. When we talk about a deterministic problem being solvable in polynomial time, we refer to P. The next chapter will define the complexity classes Randomized Time ($RTIME$) and Compliment-Randomized Polynomial Time (co-RP). These last two are for problems using randomization.

2.2 The Literature

So far, research has found algorithms capable of comparing specific types of circuits using Number Theoretic properties. There is a deterministic algorithm that can check for equality in polynomial time if the arithmetic circuits only use multiplication as operation; we also have another deterministic algorithm that checks for inequality assuming that deep number-theoretic conjectures are true if the arithmetic circuits only use multiplication as operation (Etessami, Stewart, and Yannakakis 2014). When addition and multiplication are allowed as operations, the situation becomes more difficult. For equality testing, Schönhage 1979 outlines an efficient polynomial-time algorithm that requires randomisation; this is the best we have discovered so far, and de-randomizing it will have a significant impact in many domains, particularly Polynomial Identity Testing (PIT). For deciding inequality between two SLPs with $\{+, *\}$ there is no algorithm, this problem lies in the complexity class $p^{pp^{pp^{pp}}}$ (the fourth level of the counting hierarchy) (Allender et al. 2005).

Some applications for these algorithms, or comparable ones, include Polynomial Identity Testing and working with extremely small or vast numbers. We can find many circumstances where one would need to compare extremely large or small numbers; in Dr Etessami's publication (Etessami, Stewart, and Yannakakis 2014), some examples include modelling genomic sequences or computing the maximum probability parsing for stochastic context-free grammars (SCFG). Other examples using only arithmetic circuits with addition and multiplication connect to exact numerical arithmetic in game theory for computing Nash equilibrium with three or more players.

This project will be beneficial to researchers in the field and practical Polynomial Identity Testing applications. We have more experimental data and hypotheses for possible bounds on the randomised component of the algorithm. This project will support future research and applications for massive integer comparisons in developing more accurate and efficient algorithms.

2.2.1 EquSLP and Polynomial Identity Testing

The following two problems will be defined as presented by Allender et al. 2005:

- **EquSLP** (equality straight-line program) : given a straight-line program representing an integer N , decide whether $N = 0$. This correlates to the problem of proving the equality of two arithmetic circuits that represent integers.

- **ACIT** (arithmetic circuit identity testing) : given a straight-line program representing a polynomial $f \in [X_1, \ldots, X_k]$, decide whether $f = 0$. This correlates to the problem of proving the equality of two arithmetic circuits that represent polynomials.

These are the problems we want to address. They directly link to Polynomial Identity Testing (PIT), which is determining if a polynomial is equal to the zero polynomial or whether two arithmetic circuits representing polynomials are equal (chapter 7.2.2 Arora and Barak 2009, and section 12.4 Wigderson 2019). PIT is a key issue in both complexity theory and algorithm design. It connects to many identity testing algorithms, such as primality testing algorithms. PIT was used to verify equalities between complexity classes for example $IP = PSAPCE$ and $MIP = NEXPTIME$ (chapter 4 Shpilka and Yehudayo n.d.). From the definition we have of ACIT, we can see that it is equivalent to PIT.

Since PIT is significant, much effort has gone into developing a deterministic algorithm for it. It would be a breakthrough if we could find one or make some progress towards it. There has not been much development, unfortunately. The most recent contribution is by Kabanets and Impagliazzo 2004, which demonstrated that a deterministic approach would yield circuit lower bounds on arithmetic complexity. Many randomised algorithms have been proposed so far, such as the Schwartz-Zippel algorithm (Demillo and Lipton 1978; Zippel 1979; Schwartz 1980), which uses the concept of replacing the variables in the polynomial with random values from a large enough domain, yielding a zero value if the polynomial is the zero polynomial with a high probability. The Agrawal and Biswas 2003 algorithm is a more advanced randomised algorithm that utilises the Chinese-Reminder Theorem (see Niven, Zuckerman, and Montgomery 1991) to decrease the degree of the polynomial.

These algorithms work and have a high chance of success; however, Allender et al. 2005 demonstrate that ACIT is polynomial-time equivalent to EquSLP. What is the significance of this? We can show that EquSLP is a specific case of ACIT. EquSLP is

in the co-*RP* complexity class because it can be solved using the algorithm described by Schönhage 1979, and ACIT is also in co-*RP* (Gonnet 1984; Demillo and Lipton 1978; Schwartz 1980; Zippel 1979). This result means that if we can find a deterministic algorithm for the specific case EquSLP of ACIT, we will be close to finding a deterministic algorithm for PIT and finding circuit lower bounds. Using the simpler EquSLP algorithm of Schönhage 1979 would make it easier to solve ACIT without needing to use more complex algorithms like the two listed above (Schwartz-Zippel and Agrawal-Biswas).

This project focuses on implementing the best-known algorithm for EquSLP described by Schönhage 1979, which focuses on calculating each gate of the arithmetic circuit modulo a random integer. In the next chapter, we are going to go through this method in more detail.

Chapter 3

The Algorithm

In this chapter, we are going to explain randomized algorithms, the algorithm described by Schönhage in 1979, we will touch on its weaknesses, we will look through at a worked example and finally, we will present the implementation achieved in this project using Python.

3.1 Randomized Algorithms

A *randomized algorithm* is a ”black box” that given input data and a random component, it will then proceed to do some calculations according to the input and random component and then, finally, return a result. We could fix the input data, but the algorithm might use different random components and hence deliver different results (Andrzejak 1998). Deterministic algorithms do not have a random component and hence deliver the same result on the same input. The reason for randomized algorithms is that they can be more efficient (faster, less space used or more straightforward conceptually) than deterministic ones (Andrzejak 1998). Some problems do not have deterministic solutions that can guarantee accurate results, but can have randomized algorithms capable of solving them. We have not yet found an efficient (polynomial time) deterministic algorithm for EquSLP; we only have randomized ones.

The computational complexity of a randomized algorithm can be explained through a game-theoretic perspective. We have a player represented by the algorithm and an adversary who can only choose the input data. The goal of the adversary is to find input data capable of ”foiling” the algorithm and hence ensuring we get a wrong result from it (Andrzejak 1998). The point of a randomized algorithm is to be resistant to such attacks; this is possible as the same input may use different random components

and deliver different results.

The randomized algorithm must make a decision depending on some calculations; this decision usually depends on finding a ”witness” that proves the input data posses a particular property of interest (Andrzejak 1998). The calculations involve the input data and the random component that is a member of the search space. By selecting many random components in the search space, we can, with a certain probability, find a witness if the input possesses the property. If after many trials a witness is not found, then with a certain probability, we can assume the input does not have the property (Andrzejak 1998).

Randomized algorithms need to generate randomness for them to work, but how is this done? We need a Probabilistic Turing Machine, which is a Turing Machine with a random number generator. A *Turing Machine* is an abstract machine that will manipulate symbols on an infinite tape following set rules (chapter 1, Arora and Barak 2009). Any computer is a Turing Machine. A random number generator will generate random numbers; one should not find patterns or relations between the numbers as they are random. We do not know if true randomness exists in this universe; this is still an open question, and due to this, we cannot guarantee that a random number generator is truly random (chapter 7, Arora and Barak 2009). We will discuss this limitation in section 3.5 and how to overcome this ”true randomness” problem.

3.1.1 Monte Carlo Algorithms

There are many types of randomized algorithms. The type of algorithm we want to implement is called a Monte Carlo Algorithm. ”A Monte Carlo Algorithm is an algorithm that either stops and does not answer or stops and gives a solution which is possibly not correct” (Andrzejak 1998). A crucial property of Monte Carlo Algorithms is that running it many times on the same input decreases the probability of failure at the expense of running time.

The complexity class Bounded-Error Probabilistic Polynomial Time (*BPP*) encloses the randomized algorithms with two-sided errors; this means that the algorithm, with some probability, can output ’false’ when the input has the desired property and output ’true’ when the input does not have the desired property (chapter 7, Arora and Barak 2009). The complexity classes Randomized Polynomial Time (*RP*) and Compliment-Randomized Polynomial Time (co-*RP*) enclose the randomized algorithms with one-sided error (each for each side), $RP \subseteq BPP$ and co-$RP \subseteq BPP$ (chapter

7, Arora and Barak 2009). Our algorithm is in co-*RP*, which is defined as follows:

Definition: The class Randomized Time $RTIME(t(n))$ contains every set of inputs that have the desired property L for which there is a Probabilistic Turing Machine M running in $t(n)$ time such that for an input x if

$$x \in L \implies \mathbf{Pr}[M \text{ outputs true}] = 1$$
$$x \notin L \implies \mathbf{Pr}[M \text{ outputs true}] \leq k < 1$$

where k is a constant strictly smaller than one and usually around $\frac{1}{2}$; therefore co-$RP = \bigcup_{c>0} RTIME(n^c)$ where c is a constant (chapter 7, Arora and Barak 2009).

3.2 Congruences

Let us first introduce some core definitions and properties necessary for the understanding of the algorithm:

Definition:
If m divides $a-b$ then there exists an integer q such that $a-b=mq$. "If an integer m, not zero, divides the difference $a-b$, we say that a is congruent to b modulo m and write $a \equiv b \pmod{m}$. If $a-b$ is not divisible by m, we say that a is not congruent to b modulo m, and in this case we write $a \not\equiv b \pmod{m}$." (chapter 2, Niven, Zuckerman, and Montgomery 1991).

Modular Arithmetic Properties:
Let a, b, c and d denote integers, then
(i) If $a \equiv c \pmod{m}$ and $b \equiv c \pmod{m}$, then $a \equiv b \pmod{m}$.
(ii) If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then $a+c \equiv b+d \pmod{m}$.
(iii) If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then $a*c \equiv b*d \pmod{m}$.
(chapter 2, Niven, Zuckerman, and Montgomery 1991).

Proof:
(i) $a-c=mq$ and $b-c=mp$ for integers q and p. This implies $a=mq+c$ and $b=mp+c$ hence $a-b=mq+c-mp-c=m(q-p)$. This implies that $a-b$ is a multiple of m and hence $a \equiv b \pmod{m}$.

(ii) $(a+c)-(b+d) = (a-b)+(c-d) = mq+mp = m(q+p)$ for integers q, p. This implies that $(a+c)-(b+d)$ is a multiple of m and hence $a+c \equiv b+d \pmod{m}$.
(iii) $a-b = mq$, $c-d = mp$ for integers q, p. Then $ac-bd = ac-bd+bc-bc = c(a-b)+b(c-d) = c(mq)+b(mp) = m(cq+bp)$. This implies that $ac-bd$ is a multiple of m and hence $a*c \equiv b*d \pmod{m}$.

3.3 The Algorithm Itself

We could compute the total value in binary to compare arithmetic circuits, but once again, this is very inefficient. The algorithm uses the above-stated properties of congruences recursively. We let X and Y be two arithmetic circuits and the integer value they represent throughout this project. The algorithm randomly chooses a suitable large enough integer R and computes each arithmetic operation of both circuits modulo R by repeatedly applying properties (ii) and (iii). This process ensures that the largest integer to compute is $(R-1)^2$, which is smaller and more manageable than the actual value of the arithmetic circuit. Once we have the values of both circuits modulo R, which corresponds to the value of the last gate of each circuit modulo R as arithmetic circuits are directed acyclic graphs, we can decide if they are equal or not. If we have the same result, then the algorithm outputs true for equality with high probability. If we have different results, then the algorithm always outputs false for equality. Hence, if $X = Y$, we always get true and if $X \neq Y$, we get false with high probability. This algorithm runs in polynomial time according to the combined size of both circuits that compose the input.

3.3.1 Worked Example

Let X and Y be two arithmetic circuits:

$$x_0 = 1 \qquad y_0 = 1$$
$$x_1 = x_0 + x_0 = 1 + 1 = 2 \qquad y_1 = y_0 + y_0 = 1 + 1 = 2$$
$$x_2 = x_1 * x_1 = 2 * 2 = 4 \qquad y_2 = y_1 + y_0 = 2 + 1 = 3$$
$$x_3 = x_2 * x_2 = 4 * 4 = 16 \qquad y_3 = y_2 * y_2 = 3 * 3 = 9$$
$$x_4 = x_3 + x_2 = 16 + 4 = 20$$
$$x_5 = x_4 + x_2 = 20 + 4 = 24$$

We are going to do a worked example with $R = 3$. We should obtain $X \equiv 24 \equiv 0 \pmod 3$ and $Y \equiv 9 \equiv 0 \pmod 3$.

$$x_0 \equiv 1 \pmod 3 \qquad y_0 \equiv 1 \pmod 3$$
$$x_1 \equiv x_0 + x_0 \equiv 1 + 1 \equiv 2 \pmod 3 \qquad y_1 \equiv y_0 + y_0 \equiv 1 + 1 \equiv 2 \pmod 3$$
$$x_2 \equiv x_1 * x_1 \equiv 2 * 2 \equiv 1 \pmod 3 \qquad y_2 \equiv y_1 + y_0 \equiv 2 + 1 \equiv 0 \pmod 3$$
$$x_3 \equiv x_2 * x_2 \equiv 1 * 1 \equiv 1 \pmod 3 \qquad y_3 \equiv y_2 * y_2 \equiv 0 * 0 \equiv 0 \pmod 3$$
$$x_4 \equiv x_3 + x_2 \equiv 1 + 1 \equiv 2 \pmod 3$$
$$x_5 \equiv x_4 + x_2 \equiv 2 + 1 \equiv 0 \pmod 3$$

This example is a special case, and these are the ones that concern us. We have found a random integer $R = 3$ that yields a wrong result. We know that $X = 24 \neq 9 = Y$, but some random integers would indicate otherwise (such as 3 in this example). Who are these integers? We want to find which type of random integers R would give us the same modulo R for two different integers. Following property (i), we know that $X \equiv c \pmod R$ and $Y \equiv c \pmod R$ implies that $X \equiv Y \pmod R$ and hence $(X - Y) = R * q$ for an integer q. Therefore any R that divides $X - Y$ would give the same modulo R for both X and Y. By following the example, we have $X - Y = 24 - 9 = 15$, the divisors (integers that divide into another integer exactly without leaving a remainder) of 15 are 1, 3, 5 and 15. We have found all the random integers that would make our algorithm yield a wrong result in this specific example. An observation is that the divisors of

$X - Y$ will never be bigger than $max(X,Y)$. This observation indicates that we always get the right result by picking a random integer bigger than the value of both arithmetic circuits we want to compare. Doing this is not convenient as, in theory, the arithmetic circuits we want to compare equate to gigantic integers; this would force our algorithm to select an even bigger gigantic integer that is not manageable nor efficient. What about avoiding the factors of $X - Y$? To find these factors, we would need to know what integers are represented by X and Y, and if we knew these values, we could directly decide if they are equal or not.

3.4 Schönhage 1979

In the Schönhage 1979 paper "On The Power Of Random Access Machines", we find the first depiction of the algorithm we implemented to decide equality between arithmetic circuits or straight-line programs. The paper includes a Lemma used to reinforce the algorithm's feasibility and calculate an upper bound of the probability of success.

Lemma (Schönhage 1979):
There exits a constant $c > 0$ such that for any integers $x \neq y$ generated by a straight-line program of length n with operations $\{+,*\}$, there are more than $\frac{2^{cn}}{cn}$ integers $R < 2^{cn}$ with $x \not\equiv y \pmod{R}$.

Proof (Schönhage 1979):
The number of primes $R < 2^{cn}$ not dividing $|x-y| < 2^{2^n}$ is at least $\pi(2^{cn}) - 2^n > \frac{2^{cn}}{cn}$, for all n with suitable $c > 0$.

In the proof $\pi(x)$ defines the number of prime numbers less or equal than x, it is called the Prime Number Theorem, $\lim_{x\to\infty}(\pi(x)/\frac{x}{\log(x)}) = 1$.

In the proof, we have an upper bound on the difference of two arithmetic circuits: 2^{2^n}. This upper bound represents the maximum value an arithmetic circuit can attain with a length of n gates (we are not counting the input node equal to 1 and the output node) and operations $\{+,*\}$. Each gate can only take two inputs and can either use addition or multiplication. Using multiplication will increase the value of the circuit faster than addition. Hence, we will maximise the number of multiplications. If we only use the input gate, we will multiply one n times; hence, we need at least one addition. We get the following arithmetic circuit:

$$
\begin{aligned}
&x_0 = 1\\
&x_1 = x_0 + x_0 = 1 + 1 = 2^{2^0} = 2\\
&x_2 = x_1 * x_1 = 2 * 2 = 2^{2^0} * 2^{2^0} = 2^{2^1} = 4\\
&x_3 = x_2 * x_2 = 4 * 4 = 2^{2^1} * 2^{2^1} = 2^{2^2} = 16\\
&x_4 = x_3 * x_3 = 16 * 16 = 2^{2^2} * 2^{2^2} = 2^{2^3} = 256\\
&\dots\\
&x_n = x_{n-1} * x_{n-1} = 2^{2^{n-2}} * 2^{2^{n-2}} = 2^{2^{n-1}}\\
&x_{n+1} = x_n * x_n = 2^{2^{n-1}} * 2^{2^{n-1}} = 2^{2^n}.
\end{aligned}
$$

This is the biggest value we can get in the minimum amount of gates, as getting bigger values would require more gates with addition. We see that with n gates we get $2^{2^n} > 3^{2^{n-1}} > 5^{2^{n-2}} > \dots$.

k-th Test (Schönhage 1979):
We want to find out if $X = Y$ when X and Y are any integers generated by a straight-line program of size n with operations $\{+,*\}$. The k-th test consists of randomly choosing v_k many values of R, independently and equally distributed in $1 \leq R \leq 2^{cn}$. For each of the v_k random values we execute both straight-line programs mod R. If we get $X \not\equiv Y \pmod{R}$ with any of the selected R then we are sure that $X \neq Y$. Otherwise, we assume that $X = Y$. Using the previous Lemma, and for any $\varepsilon > 0$, we get an upper bound on the probability of failure:

$$(1 - \frac{1}{cn})^{v_k} < \exp(\frac{-v_k}{cn}) < \frac{\varepsilon}{2^k}.$$

If we choose $v_k = \lceil cn(k + \log \frac{1}{\varepsilon}) \rceil$, then the probability of success is greater than $\prod_k (1 - \frac{\varepsilon}{2^k}) > 1 - \varepsilon$ and the running time only grows polynomially in $\log \frac{1}{\varepsilon}$.

From the Lemma we get that there exits a constant $c > 0$, we can see that $c > 2$ suffices, such that when the SLP has size n the probability of failure is bounded by $(1 - \frac{1}{cn})$. This is because there are more than $\frac{2^{cn}}{cn}$ integers $1 \leq R < 2^{cn}$ with $X \not\equiv Y \pmod{R}$ where $X \neq Y$; hence the probability we pick an integer R such that $X \equiv Y \pmod{R}$ is $(1 - \frac{2^{cn}}{cn} * \frac{1}{2^{cn}})$. If n goes to infinity we get a probability of failure equal to one. The k-th Test helps us overcome this, giving us an upper bound of $(1 - \frac{1}{cn})^{v_k}$.

Setting $v_k = cn * d$, we get an approximated probability of failure of $\exp(\frac{-cn*d}{cn}) = \exp(-d)$. This is great, as we can choose a d that will give us a desired amount. For example, we want an error of less than 0.1%, we solve for d and get $d = 6.90775\ldots$. We can make the upper bound on the probability of failure arbitrarily low by running the algorithm more times.

3.4.1 Limitations

Schönhage 1979 gives us an upper bound on the probability of failure and a method to decrease it as low as we need to; this is good, but in many cases, this is highly wasteful. For example, setting c to a reasonable minimum ($c = 2$) and choosing $d = 7$ (so we get probabilities smaller than 0.1%) tells us we have to run the algorithm $14 * n$ times to get a minimal error, but n can be a massive number. If we only run it once, we know the upper bound on the probability of failure will go to 1 as $n \to \infty$. With Schönhage's Lemma, we can only rely on running the algorithm multiple times to ensure low probabilities.

Let us look at the previous worked example. We have two SLPs X and Y. We can make them the same size by adding a padding of $*1$ using $y_0 = 1$ to Y. Assume both circuits are of the same size 4. If we pretend we do not know what the difference is, we can only bound it $|X - Y| < 2^{2^4}$. Looking at the proof we realize that the minimum c we can choose is $c = 2$, this gives from the Lemma and Proof that there are more than $\frac{2^{2*4}}{2*4} = \frac{2^8}{8} = 32$ integers $R < 2^8 = 256$ with $X \not\equiv Y \pmod{R}$. This yields the following upper bound on the probability of failure for only one run: $(1 - \frac{1}{2*4}) = 0.875$. In reality we know that $|X - Y| = 15$ and if we choose R in $1 \leq R < 2^{2*4} = 256$ we get a true probability of $\frac{4}{256} = 0.015625$, as 15 has 4 divisors. But we can do even better and realize that 1 divides all integers, and pick R in $1 < R < 2^{2*4} = 256$, and hence get a true probability of $\frac{3}{255} = 0.01176\ldots$. This is a specific case but, we can hypothesize that the majority of cases are like this one. This will be studied further in the next chapter.

Finally, the range for R we get from the k-th test is too vast ($1 \leq R < 2^{cn}$). We do not have any refinement; limiting R to sub-ranges could greatly impact the probability of failure. Schönhage uses the Prime Number Theorem to prove the Lemma showing that there are enough primes to satisfy the inequation. What about composite numbers (non-prime)? Finding big primes for R is inefficient, and maybe picking random integers can do just as well. We will look at this in the next chapter.

3.5 The Code

We are dealing with huge integers; thus, we will need a programming language that can handle them without rounding, as we will require exact values for comparison. Python can handle any size integer without needing to specify it explicitly.

To execute the algorithm, we need a random number generator. In modern computers, we define these generators as pseudo-random number generators. We can not guarantee true randomness, so we simulate it; this is why we have ”pseudo”. A good pseudo-random number generator should be indistinguishable from a true random number generator. Imagine that we want to simulate a fair coin, but we only have access to a biased toss ($\mathbf{Pr}[\text{Heads}] = \rho \neq \frac{1}{2}$). Von-Neumann proved that a fair coin (i.e. $\mathbf{Pr}[\text{Heads}] = \frac{1}{2}$) can be simulated by a Probabilistic Turing Machine that can repeatedly toss a ρ-coin (chapter 7, Arora and Barak 2009). This technique is adaptable to any probability we need.

The algorithm uses the ”randint()” function to generate random integers using seeds as a base from a specified range. If we use the same seed, ”randint()” will deliver the same sequence of numbers. If a seed is not specified, Python will use the current system time in milliseconds.

The algorithm has many other design choices that will be cover in the following chapter in section 4.6. These design choices come from the results of experimental research.

```python
#!/usr/bin/env python3

import math
from random import randint

def EquSLP(X,Y):

    """
    EquSLP is used to decide if two Straight Line Programs are
    equal by simply choosing a random "large enough" integer, R,
    and calculate each arithmetic operation of each SLP modulo R.
    If both SLPs have the same modulo R, then they might be equal.
    This process is run many times with different R values,
    if it is always equal, we can conclude that both SLPs are equal
    with high probability and return True; otherwise, return False.

    Input: two SLPs X and Y. All SLPs start with 1.
    If X is x0 = 1, x1 = x0 + x0 and x2 = x1 * x1
    then X = [1, (0,0,"+"), (1,1,"*")].
    """

    # calculate an appropiate amount of bits for the random
    # integer R and runs depending on the size of the input
    if max(len(X), len(Y)) - 2 < 900:
        runs = 30
        bits = 30
    else:
        runs = math.ceil(math.sqrt(max(len(X), len(Y)) - 2))
        bits = math.ceil(math.sqrt(max(len(X), len(Y)) - 2))


    for _ in range(runs):

        # generate random integer
        R = randint(2**(bits - 1) , 2**(bits))

        # first gate is always 1
        modX = [1] # list of all the gates in X modulo R
        modY = [1] # list of all the gates in Y modulo R


        # calculate the modulo R of each gate in X
        for i in range(1,len(X)):

            if X[i][2] == "*":

```

```
47                modX.append((modX[X[i][0]] * modX[X[i][1]]) % R)
48
49            elif X[i][2] == "+":
50
51                 modX.append((modX[X[i][0]] + modX[X[i][1]]) % R)
52
53
54        # calculate the modulo R of each gate in Y
55        for i in range(1,len(Y)):
56
57            if Y[i][2] == "*":
58
59                modY.append((modY[Y[i][0]] * modY[Y[i][1]]) % R)
60
61            elif Y[i][2] == "+":
62
63                 modY.append((modY[Y[i][0]] + modY[Y[i][1]]) % R)
64
65
66        # the last gate modulo R is equal to the whole SLP modulo R
67        if modX[-1] != modY[-1]:
68            return False
69
70    return True
```

Chapter 4

Experimental Research

This chapter will discuss the experiments and tests carried out during this project, the results found and possible hypotheses to justify them, and finally, how the experimental research shaped the algorithm and its performance.

4.1 The Layout

We will become an attacker and design different arithmetic circuits to attain high probabilities of failure. We want to find the vulnerabilities of the algorithm so we can make it stronger and better performing with any input.

The key factor of the algorithm is the random component. Choosing an adequate one would give us the right results, but how can we know if it is right? An attacker has full control over the input; the algorithm needs to use the information in the input to decide what type of random integer is needed to find the right answer. We saw in chapter three that the maximum value an arithmetic circuit can attain with n gates is 2^{2^n}. This value gives us an upper bound for the difference between the two SLPs we compare. We can use this bound to decide a range to select from the random component. In Schönhage's k-th test we choose R in $1 \leq R \leq 2^{cn}$ for an adequate integer $c > 0$. We see that if $R = 1$, any arithmetic circuit mod R would be equal to zero, indicating that any two SLPs are equal. We can hence eliminate this option. If we choose $2^{2^n} \leq R$ where n is the size of the biggest SLP we want to compare, we always get the right answer, but this is not efficient as we want to avoid doing computations with huge integers. Hence we can choose $c > 0$ such that we get the following new range for R: $1 < R \leq 2^{cn} \leq 2^{2^n}$. Even if we limit c to $c = 2$, this range can be too vast; we can randomly select $R = 2$ indicating that any two even SLPs are equal, or

$R = 2^{2^n} - 1$ which can be a huge integer and slow down run-time drastically.

An intuition is to separate this range into smaller ranges so we can have a fine-tuning on the random component and maybe guarantee better probabilities. This intuition comes from the fact that divisors of an integer are not uniformly distributed. An integer n cannot have divisors bigger than $\frac{n}{2}$ excluding itself as a divisor. Let a be a divisor such that $\frac{n}{2} < a < n$ and $a * b = n$; if $b = 1$ then $a = n$, a contradiction; what about $b \geq 2$ then $a * b > \frac{n}{2} * 2 > n$, another contradiction. Another fact is that all divisors come in pairs and one of the two divisors in each pair is less than $\sqrt{n}$. Let a and b be two divisors such that $a, b > \sqrt{n}$ then $a * b > \sqrt{n} * \sqrt{n} = n$, a contradiction; hence divisors come in pairs such that $a < \sqrt{n} < b$ unless $\sqrt{n}$ is an integer. These two facts demonstrate that the distribution of divisors of an integer is skewed, and integer n will have half of its divisors (excluding 1 and n) in $[1, \sqrt{n}]$ and the other half (excluding 1 and n) in $[\sqrt{n}, \frac{n}{2}]$. As the random number we choose grows, the probability of failure approaches zero because as R gets bigger, it will eventually surpass 2^{2^n}, where n is the size of the longest circuit, and therefore the answer obtained by modular arithmetic will always be the correct answer. We would like to know how fast the probability of failure decreases and how close to 2^{2^n} we have to go to attain very low probabilities.

We split the range into smaller sub-ranges of integers that are represented by the same amount of bits. "A positive integer n has b bits when $2^{b-1} \leq n \leq 2^b$–1" (Regan 2012). Sub-ranges are a natural division of the range, allowing us to fine-tune R and look for a relationship between the input, the number of bits of R, and the probability of failure. We get the following sub-ranges:

$$bits = 1 \implies b_1 = [2^0, 2^1 - 1] = [1,1] \implies |b_1| = 1$$
$$bits = 2 \implies b_2 = [2^1, 2^2 - 1] = [2,3] \implies |b_2| = 2$$
$$bits = 3 \implies b_3 = [2^2, 2^3 - 1] = [4,7] \implies |b_3| = 4$$
$$bits = 4 \implies b_4 = [2^3, 2^4 - 1] = [8,15] \implies |b_4| = 8$$
$$bits = 5 \implies b_5 = [2^4, 2^5 - 1] = [16,31] \implies |b_5| = 16$$
$$bits = 6 \implies b_6 = [2^5, 2^6 - 1] = [32,63] \implies |b_6| = 32$$
$$bits = 7 \implies b_7 = [2^6, 2^7 - 1] = [64,127] \implies |b_7| = 64$$
$$bits = 8 \implies b_8 = [2^7, 2^8 - 1] = [128,255] \implies |b_8| = 128$$
$$\dots$$
$$bits = 2^n \implies b_{2^n} = [2^{2^{n-1}}, 2^{2^n} - 1] \implies |b_{2^n}| = 2^{2^n} - 2^{2^{n-1}} = 2|b_{2^{n-1}}|$$

4.2 Exploratory Tests

This section consists of exploratory tests. These are ideas or intuitions of different types of circuits we could compare, trying to maximise the value of each circuit using few gates and seeing what we get. The data collected is composed of input size, number of bits of the random component and probability of failure. The code used to generate the data and analyse it is in appendix B.1 and B.2; many more graphs were generated and found in appendix A.1 and A.2.

All the tests presented in this section followed the same format. We decided to calculate the probability of failure by picking at random one thousand integers in each of $b_8, b_9, \dots, b_{32}$ and count the number of times the algorithm indicated the arithmetic circuits were equal. All the pairs of arithmetic circuits are not to be equal. We tested random components from height bits to thirty-two bits as these are comfortable ranges (i.e. less than 8 bits numbers are too small, and more than 32 bits was not necessary as we will see probabilities will reach values close to zero very fast).

4.2.1 $a + b * c^{2^{1,\dots,n}} \stackrel{?}{=} a + b * c^{2^{1,\dots,n-1}}$

The point of the algorithm is to compare huge integers represented by SLPs. Hence we will start comparing circuits that represent $a + b * c^{2^n}$ with different n. This test

compares

$$
\begin{aligned}
a+b*c^{2^{64}} &\stackrel{?}{=} \{a+b*c^{2^{63}},\ a+b*c^{2^{62}},\ a+b*c^{2^{61}},\ \ldots,\ a+b*c^{2^1}\}\\
a+b*c^{2^{63}} &\stackrel{?}{=} \{a+b*c^{2^{62}},\ a+b*c^{2^{61}},\ a+b*c^{2^{60}},\ \ldots,\ a+b*c^{2^1}\}\\
a+b*c^{2^{62}} &\stackrel{?}{=} \{a+b*c^{2^{61}},\ a+b*c^{2^{60}},\ a+b*c^{2^{59}},\ \ldots,\ a+b*c^{2^1}\}\\
&\ldots\\
a+b*c^{2^2} &\stackrel{?}{=} a+b*c^{2^1}
\end{aligned}
$$

We went beyond $n = 64$, and b_{32}, but the results were very similar, and the time the test took increased. With this set of parameters, we can understand how the probability changes depending on the size in bits of the random component. We tested with many combinations for a, b and c, but we got very similar results in each case. Usually, doing $0+1*p^{2^n}$ for p a prime gave us higher probabilities than other combinations.

Let us look more in detail at an example that is the most intuitive one we can think of: $0+1*2^{2^{64}}$. The results are in figure 4.1. This graph reflects what we will usually find with any other values for a, b and c. Each pair of SLPs we compared (for example $2^{2^{64}} \stackrel{?}{=} 2^{2^4}$) has a grey point for 8 bits, 9 bits, ... indicating the probability of failure if we picked R in the ranges $b_8, b_9, \ldots$. We see in figure 4.1 how the probabilities decrease as the number of bits of R increases. We start with a worse probability around 0.8 at 8 bits and end with a worse probability around 0 at 32 bits. The red curve in figure 4.1 represents the mean probability for each range $(b_8, b_9, \ldots)$. The mean probability decreases as well as the variance and standard deviation. All of these are clear signs that the more bits, the better.

Why is this happening? Two reasons: first, we saw that divisors are not evenly distributed and are rarer the closer we get to the value they divide; second, each time we increase one bit, the range we pick R from doubles in size. We see that in this specific example, by picking $2^{32-1} \leq R \leq 2^{32}$–1, we do not need to run the algorithm many times; we get excellent results already. Schönhage would have indicated $1 \leq R < 2^{c*64}$ and a probability lower than $1 - \frac{1}{c*64}$.

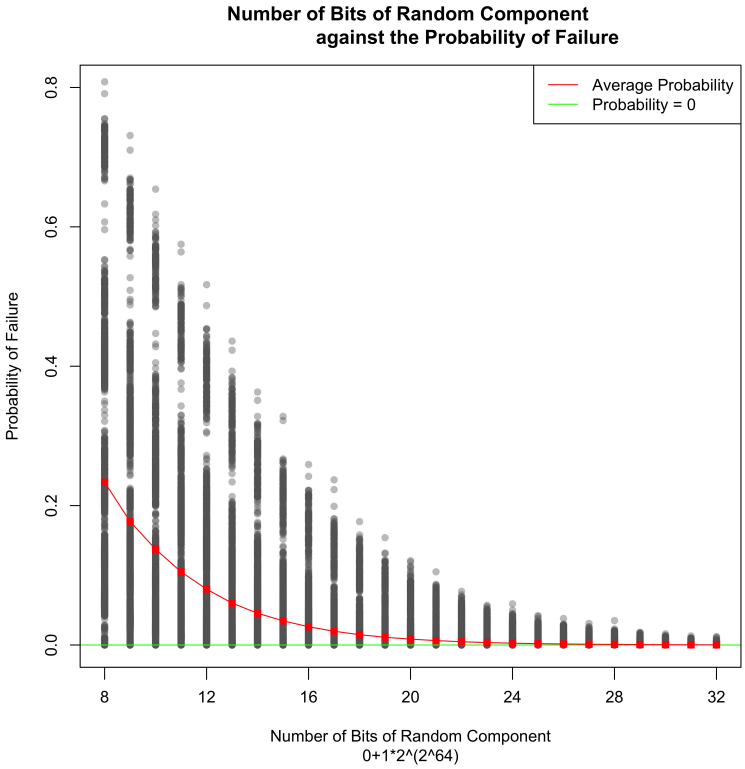


Figure 4.1: $2^{2^{1,\ldots,64}} \stackrel{?}{=} 2^{2^{1,\ldots,63}}$.

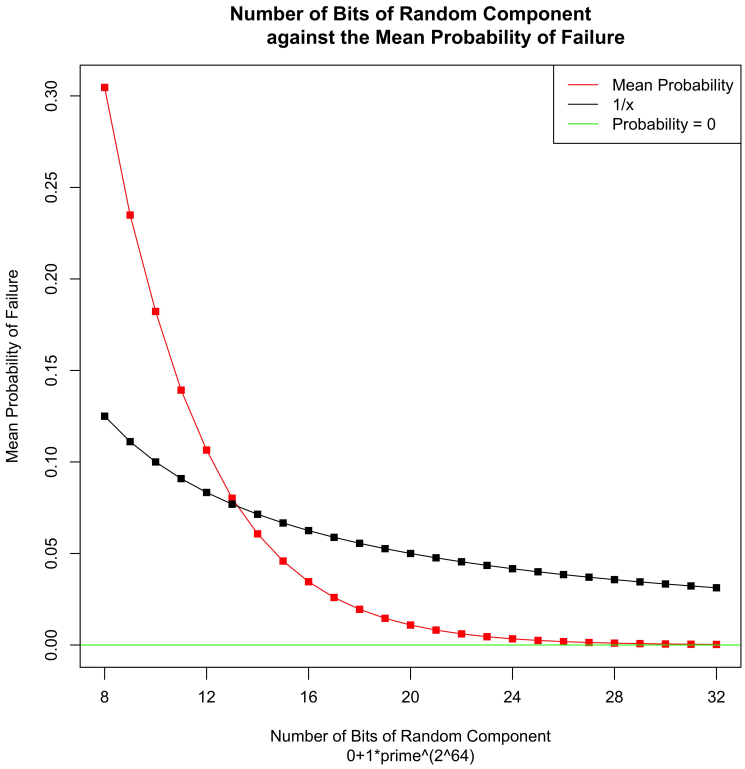


Figure 4.2: Mean Probability of failure: $p^{2^{1,\ldots,64}} \stackrel{?}{=} p^{2^{1,\ldots,63}}$, for all primes $p < 100$.

We want to understand how fast the probability decreases depending on the number of bits of R. We have run the same test, $0+1*\text{prime}^{2^{64}}$, for all primes smaller than 100 and then calculated the mean probability of failure across all the results (figure 4.2). It would be great to find a bound on the mean probability of failure. The most straightforward function we can think of is $\frac{1}{x}$ where x is the number of bits of R. We see in figure 4.2 that as the amount of bits grows, $\frac{1}{x}$ does bound the curve. We can hypothesize that a function that bounds the mean probability of failure across all possible arithmetic circuits exists and tends to zero as the number of bits of R goes to infinity.

4.2.2 $a+b*c^{2^{1,\ldots,n}} \stackrel{?}{=} x+y*z^{2^{1,\ldots,n}}$

The first test yielded some very interesting results, but we would like to compare even more diverse circuits. We compare $a+b*c^{2^n}$ to $x+y*z^{2^n}$ for any a,b,c,x,y and z with different n. The previous test is a special case of this one if we let $a=x, b=y$ and $c=z$. This test compares

$$
\begin{aligned}
a+b*c^{2^{64}} &\stackrel{?}{=} \{x+y*z^{2^{64}},\ x+y*z^{2^{63}},\ x+y*z^{2^{62}},\ \ldots,\ x+y*z^{2^1}\}\\
a+b*c^{2^{63}} &\stackrel{?}{=} \{x+y*z^{2^{64}},\ x+y*z^{2^{63}},\ x+y*z^{2^{62}},\ \ldots,\ x+y*z^{2^1}\}\\
a+b*c^{2^{62}} &\stackrel{?}{=} \{x+y*z^{2^{64}},\ x+y*z^{2^{63}},\ x+y*z^{2^{62}},\ \ldots,\ x+y*z^{2^1}\}\\
&\ldots\\
a+b*c^{2^1} &\stackrel{?}{=} \{x+y*z^{2^{64}},\ x+y*z^{2^{63}},\ x+y*z^{2^{62}},\ \ldots,\ x+y*z^{2^1}\}
\end{aligned}
$$

Once again we limit the search space to $n=64$ and $b_8, b_9, \ldots, b_{32}$. As we saw earlier results are very similar; changing the values of all the variables would yield very similar results every time. We compared many combinations of primes such as $2^{2^{1,\ldots,n}} \stackrel{?}{=} 3^{2^{1,\ldots,n}}$, and the results are in figure 4.3. We see in figure 4.3 that the same observations are made from the previous test but the probabilities are smaller; even for 8 bits we get worse probabilities around 0.05. This smaller probabilities are found across the majority of combinations we tried, especially while comparing different primes with each other, such as $p^{2^{1,\ldots,n}} \stackrel{?}{=} q^{2^{1,\ldots,n}}$ for p and q primes ($p \neq q$).

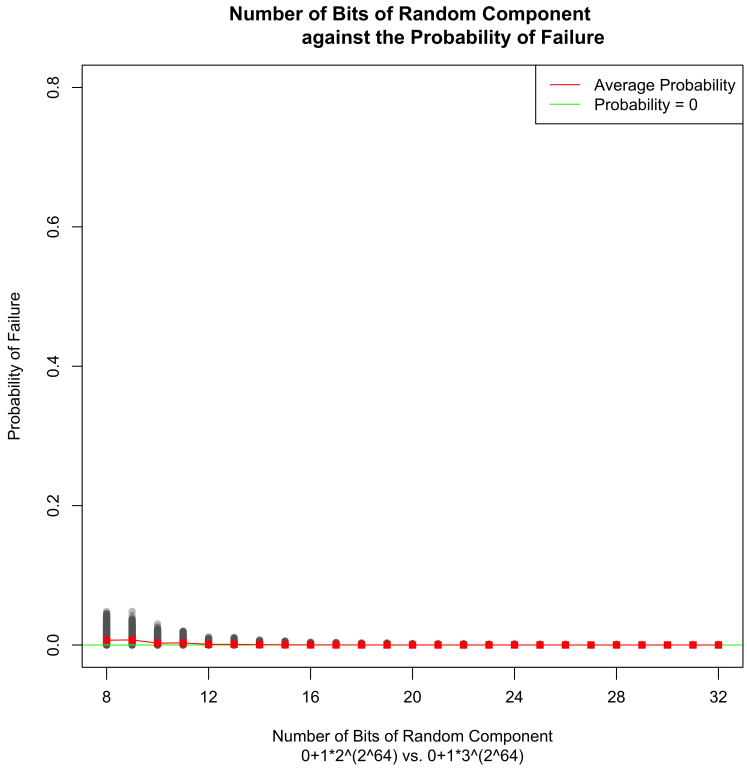


Figure 4.3: $2^{2^{1,\ldots,64}} \stackrel{?}{=} 3^{2^{1,\ldots,64}}$.

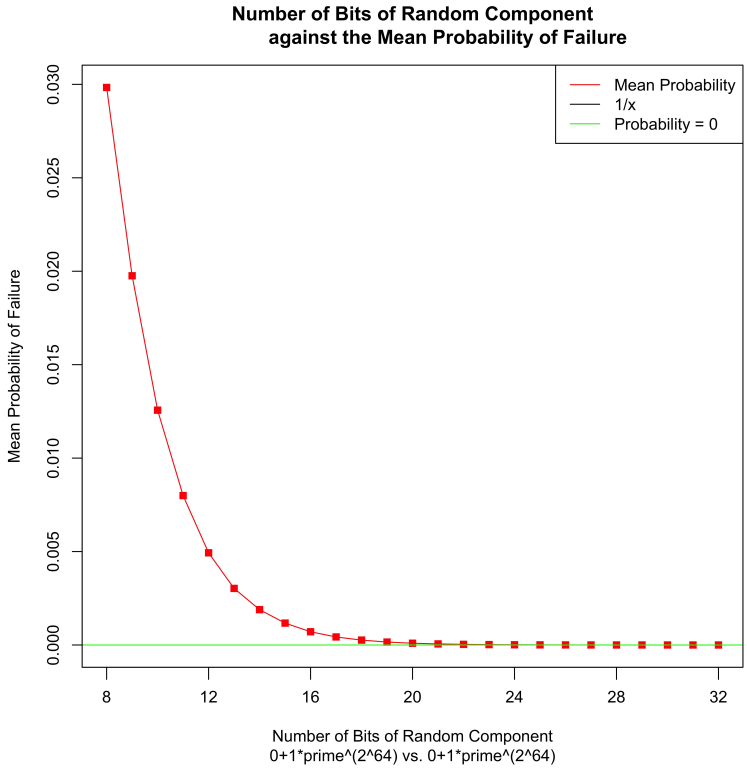


Figure 4.4: $p^{2^{1,\ldots,64}} \stackrel{?}{=} q^{2^{1,\ldots,64}}$, for all primes $p, q \leq 11, p \neq q$.

We gather the data from $p^{2^{1,\ldots,n}} \stackrel{?}{=} q^{2^{1,\ldots,n}}$ where $(p,q) \in \{(2,3),(2,5),(2,7),(2,11),(3,5),(3,7),(3,11),(5,7),(5,11),(7,11)\}$ and calculated the mean probability of failure. The results are in figure 4.4. We included the function $\frac{1}{x}$, but the mean probabilities are so small that the function doe not appear in the graph.

4.3 All Divisors

One would think that very close numbers will be the more challenging ones, as they are very similar, and we can not compute them explicitly. In reality, the actual values of the arithmetic circuits are irrelevant to the probability of failure. What matters is the value of the difference, $|X-Y|$. If we have, for example, X and $Y = X+1$ then the difference is $|X-Y| = |X-(X+1)| = 1$, and 1 only has as factors itself, so any other random integer will give correct results. The issue is when the difference between X and Y is large enough to have many divisors that can trick the algorithm. One can think to compare the size of the circuits; if they are very different in size, maybe $|X-Y|$ will be big. This intuition is not right; the size of the circuits do not reflect if the difference is big. Here we consider that an attacker can create any circuit, and we do not examine in detail the SLP to find patterns. We want a systematic approach. Circuits that will give us a huge difference ($|X-Y|$) can be represented using different size circuits or same size circuits. The size of an arithmetic circuit can be deceiving, as an attacker can do padding of $*1$ to make them seem they are the same size. We could also judge them by the number of additions and multiplications, but once again, this is deceiving, as having a lot of $*1$ does not change the value and will trick us into thinking there is much multiplication, making the value of the circuit potentially vast. One can recognise $*1$ as the only one is the input node, but then an attacker can find other ways to slow down the growth and use more gates, and hence end up with two arithmetic circuits with a huge difference in value and same size. Hence, we will ignore the difference in the size of arithmetic circuits as this is not a consistent evaluation metric.

It is not a general rule that a big difference in value between X and Y will make it harder for the algorithm, but there is a higher chance that having an enormous difference ($|X-Y|$) could lead to numbers with more divisors. We can see best-case scenarios and worst-case scenarios. If $|X-Y|$ equates to a prime number, then the only two cases of failure will be if $R = 1$ or $R = |X-Y|$, but we eliminated the option of $R = 1$; hence there is only one chance of failing. This is the best-case scenario. The worst-case scenario is when the difference is an integer with many divisors that could

easily trick the algorithm. Ramanujan defined Highly Composite Numbers (HCN) as integers that have more divisors than any smaller integer than them (Ramanujan 1915). Hence if $|X - Y|$ is a Highly Composite Number is the worst-case scenario. We study HCN in the next section.

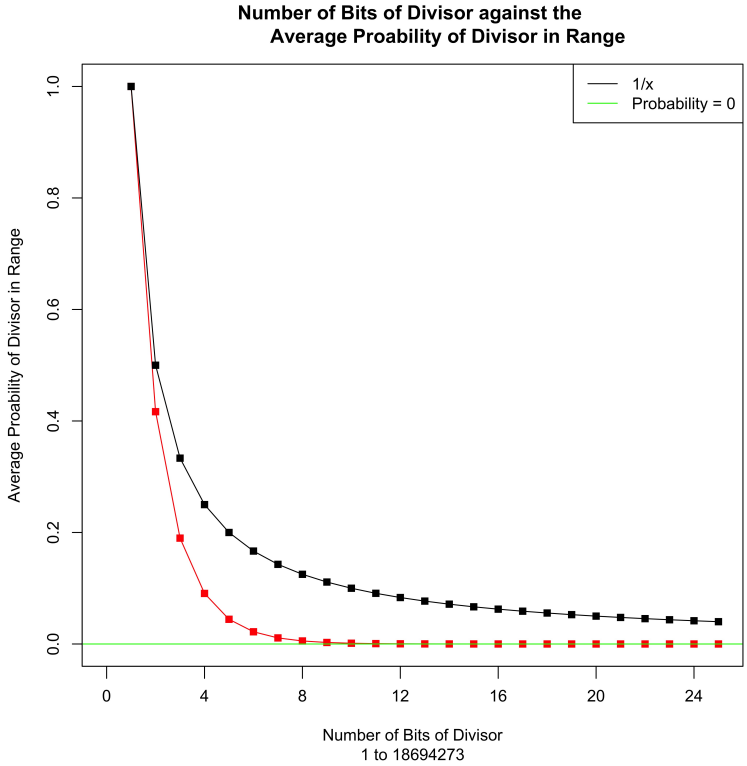


Figure 4.5: Mean Distribution of divisors from 1 to 18694273.

What truly matters is the difference $|X - Y|$ and its divisors; hence instead of trying many different SLPs, we can find the distribution of the divisors of an integer. We have tested all integers from 1 to 18694273. The code is in appendix B.3. For each integer we tested, we found all its divisors, counted how many are in each b_x where x goes from 1 to the bit size of the integer we are testing, and calculated the probability of picking a divisor in b_x (i.e. the number of divisors in b_x divided by the size of b_x). We stopped at 18694273 as the data set was already too big and hard to analyse. We ended up processing more than 300 million data points. Even if the data set sounds big, we are not grasping the infinity of numbers. We can use figure 4.5 as a very rough estimate. We can hypothesise that the mean probability of picking a divisor of an integer in b_x as $x \to \infty$ goes to zero very fast. We added the function $\frac{1}{x}$ to the graph; we see how the mean probability is bounded all the way and goes to zero faster than $\frac{1}{x}$. We can hypothesise that a function similar to $\frac{1}{x}$ bounds the mean probability of picking

a divisor of an integer in b_x as $x \to \infty$.

4.4 Highly Composite Numbers

The worst case scenario is when the difference between the two SLPs we are comparing has a lot of divisors. Let $d(n)$ be the divisor function, such as $d(n)$ gives us the total amount of divisors of n. Ramanujan 1915 defines Highly Composite Numbers (HCN) as integers that have more divisors than any smaller integer; formally n is a HCN if $d(n) > d(m)$ for any $m < n$. Ramanujan 1915 also defines the Superior Highly Composite Numbers (SHC), which is a more restricted version of HCN. A positive integer n is SHC if there exists an $\varepsilon > 0$ such that $\frac{d(n)}{n^\varepsilon} \geq \frac{d(k)}{k^\varepsilon}$ for all integers $k > 1$.

We did the same test as in section 4.3 but with the first 39 Superior Highly Composite Numbers (Sloane 2021). The code is in appendix B.3. We found all the divisors of each SHC and calculated the probability of choosing a divisor in the ranges b_x. The results are in figure 4.6. We can see that we get enormous probabilities for the first 20 bits, but then it rapidly converges to zero faster than $\frac{1}{x}$, where x is the number of bits. $\frac{1}{x}$ bounds all the probabilities after $x = 15$. For x in $[2,20]$, the variance is massive and we get probabilities of 1 for certain SHC. Once past the first 20 bits, the variance goes close to zero, the same for the probabilities. We stopped at the 39th as the integers were getting too big and the computer had difficulties. The 39th SHC is 6929234558912696097204912320919489916800 0 (Sloane 2021), it has 41 digits and is in b_{136}, meaning it is in the range $[2^{135}, 2^{136} - 1]$.

Unfortunately, we do not know what happens as we test the infinite amount of SHC numbers. Nevertheless, we can hypothesize that we will get a graph with a similar shape; very high probabilities at first and then after a turning point, all probabilities close to zero. The turning point we have in our data set is around 30 bits. Intuitively we can think that this turning point will grow as the numbers tested grow too.

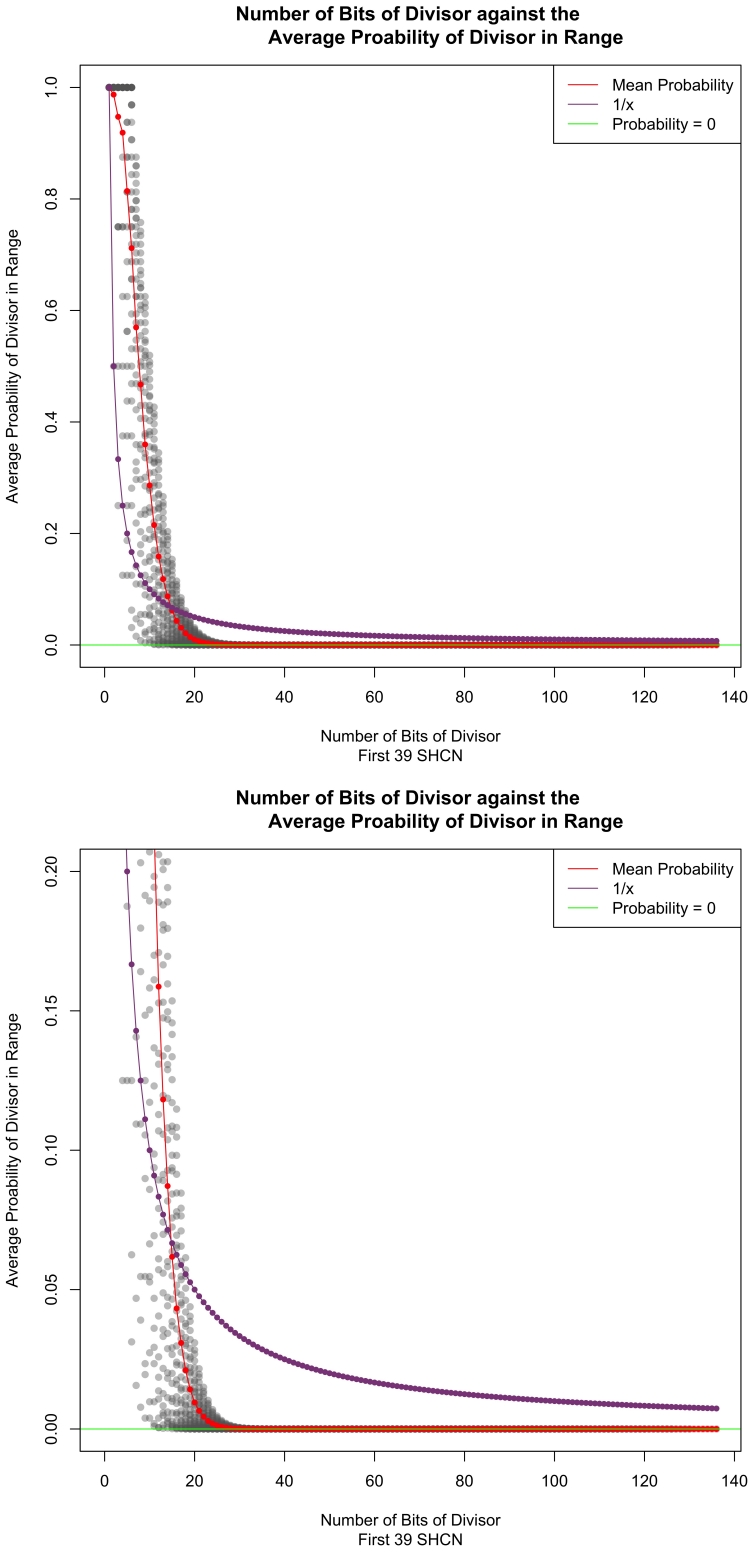


Figure 4.6: First 39 Superior Highly Composite Numbers.

4.5 The Algorithm Performance

We have tested the algorithm run-time in seconds. The code is in appendix B.4.

In the first test, we fixed the input and let the random component go from 1 bit to 512 bits. We tested every bit value one thousand times and took the average. We see the results in figure 4.7. We see an increase in run-time as the number of bits increases; we found that the data follows an approximated linear function: seconds $=$ $6.617e-0.5+5.034e-07*$bits. Even if there is an increase in run-time, the algorithm is high-speed and efficient.

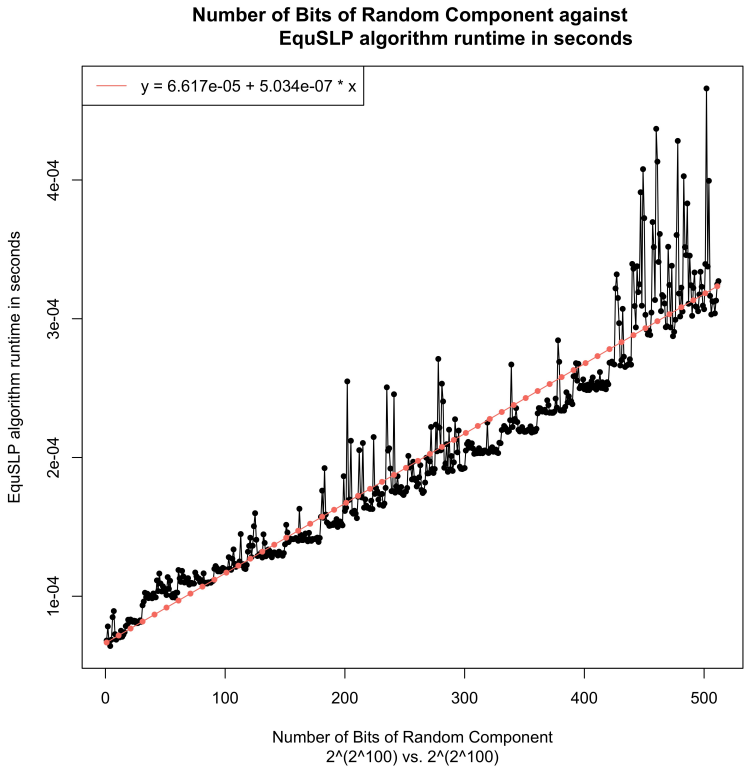


Figure 4.7: EquSLP algorithm performance when bit size of R goes from 1 to 512 bits.

In the second test, we fixed the bit size of the random integer and let the input size go from 0 gates to 500 gates. We kept Y equal to the input node having a size of 0, and we let X grow from the input node, one gate at a time. We tested every pair of SLPs one thousand times and took the average. We see the results in figure 4.8. We see an increase in run-time as the input size increases; we found that the data follows an approximated linear function: seconds $=3.625e-0.6+3.217e-07*$input size. Even if there is an increase in run-time, the algorithm is high-speed and efficient.

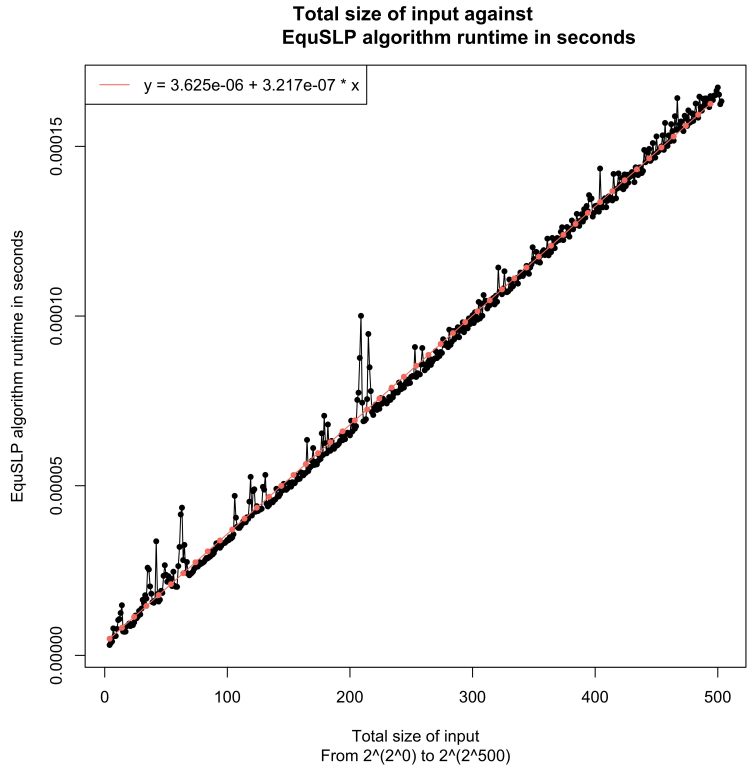


Figure 4.8: EquSLP algorithm performance when input size goes from 0 to 500 gates.

4.6 The Algorithm Design

When designing the algorithm, we need to consider both situations, when the SLPs are equal and not. If two SLPs X and Y are not equal we need to find a ”witness” R such that $X \not\equiv Y \pmod{R}$. If they are equal, we have $X \equiv Y \pmod{R}$ for any R. To find a witness, we might need to run the algorithm many times, but if there is no witness, we do not want to compromise performance by running it too many times and finding out that they are equal. We need a balance. The algorithm we programmed stops as soon as we find a witness, avoiding unnecessary runs. The number of runs can be arbitrary, and the more runs, the better results we can guarantee. We saw in the previous section that the algorithm performs very well, and all the runs we did were always below one millisecond for an average computer, but the run time increases, so we do not want to abuse this fact. We decided to do the same amount of runs as the number of bits of R, for convenience, and let the number of runs grow as the input grows. This intuition is obvious; as the input grows, the potential values of the SLPs grow as well, and they could have more and bigger divisors, so more runs guarantee low probabilities of failure.

We want to build an algorithm that decides what size is suitable for R depending on the input. In section 4.3, we explained that we could not rely on the difference in size between the SLPs, but we can use the size of the biggest one to bound the difference ($|X - Y| \leq 2^{2^n}$). We will call n the size of the biggest SLP we are comparing (n is the number of gates). We want to relate n to how many bits we use ($bits$). Across all tests, we see how the probability decreases abruptly until around 30 bits and then stays close to zero. As the number of bits of the random component grows, the probability of failure decreases. If we relate n and $bits$ linearly, $bits$ will increase faster than we need to. In the algorithm design, we want to simulate this abrupt decrease up to 30 bits.

We cannot fix the number of bits of R to a specific integer x, as an attacker can create a massive arithmetic circuit that is divisible by all the integers in b_x, guaranteeing the algorithm to fail. We need to relate n (input size) and $bits$ by a function. We don't want this function too grow to fast, so we settled for $\lceil bits = \sqrt{n} \rceil$. This function is great as it grows slowly, but not too slow, as $log()$. The only problem is that with small n, we get tiny $bits$, smaller than 30 bits. Hence we ensure that $bits$ is at minimum equal to 30. What about fixing the number of bits? If the input has less than 900 gates ($\sqrt{900} = 30$), the value of $bits$ is fixed, but 900 gates are not enough to generate an integer that has as divisors the majority of elements in b_{30}. We have tested the algorithm and the results were always as expected.

Finally, another idea that could have been implemented, but avoided for simplicity, is to increase the number of bits each run to get even better results.

Chapter 5

Conclusions

This project took us through the journey of discovering the algorithm described by Schönhage 1979, analysing it, understanding its weaknesses, implementing it, and testing it.

In this study, we examined how to compare arithmetic circuits or straight-line programs (SLP) as defined in chapter 1 (The Introduction) to determine equality between them (EquSLP). We went through the literature regarding comparing SLPs and defined the problems ACIT and EquSLP. Many applications exist for such problems and relate to deep number-theoretic fields with active research such as Polynomial Identity Testing (PIT) or simply comparing extremely large or small numbers (see chapter 2 (Background)). We introduced the algorithm studied in chapter 3 (The algorithm) and found when the algorithm fails (if R divides $|X-Y|$). Schönhage 1979 gave us a lemma and proof used to calculate the upper bound on the probability of failure: $(1-\frac{1}{cn})^{v_k}$ where v_k is the number of times we run the algorithm, n is the size of the biggest SLP we are comparing, and $c > 0$ is a constant. This upper bound was critiqued and found to be wasteful. In chapter 4 (Experimental Research), we demonstrated through a set of tests that the probability of failure is smaller than expected and decreases as R increases. We tested big sets of numbers and found that the probability of failure generally decreases very fast up to a turning point from which it gets very close to zero. Our datasets had a turning point of around 30 bits. These are excellent results, meaning that we do not have to pick huge random integers or run the algorithm many times to get outstanding results. Section 4.3 of Chapter 4 hypothesises that a function that bounds the mean probability of failure across all integers must exist. We see in our tests that for relatively small integers (compared to infinity) $\frac{1}{x}$ bounds the mean probability of failure for all integers in the range we tested; in reality, an actual bound

would decrease slower than $\frac{1}{x}$. These experimental results shaped the final algorithm design, found at the end of chapter 3.

5.1 Future Work

The scope of this project was to understand the algorithm and the probability of failure experimentally. Future work would be to try to prove or reject the hypothesis we made mathematically. During our research, we found many papers that relate to the distribution of divisors of integers; this has a deep connection to our problem and could be the way to finding a bound on the mean probability of failure and proving it. Such papers relate to the number of integers smaller than a given x that have at least one divisor in the range $(y, 2 * y]$ where $y < x$ (Ford 2008; Besicovitch 1935; Erdõs 1960; Erdõs 1935). This fits our research as the range we pick R from is $[2^{b-1}, 2^b - 1]$. The papers showed that the average number of integers smaller than x satisfying this property decreases and goes to zero as x and y increases and proved many bounds on this value.

Bibliography

Aaronson, Scott (2021). "Who Can Name the Bigger Number?" In: (), p. 18. URL: https://www.scottaaronson.com/writings/bignumbers.html (visited on 08/06/2021).

Agrawal, Manindra and Somenath Biswas (July 2003). "Primality and identity testing via Chinese remaindering". In: *Journal of the ACM* 50.4, pp. 429–443. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/792538.792540. URL: https://dl.acm.org/doi/10.1145/792538.792540 (visited on 04/30/2021).

Allender, Eric et al. (2005). "On the Complexity of Numerical Analysis". In: *Electronic Colloquium on Computational Complexity* 37.1, pp. 1–10. ISSN: 1433-8092.

Andrzejak, Artur (1998). "Introduction to randomized algorithms". In: *Lectures on Proof Verification and Approximation Algorithms*. Ed. by Ernst W. Mayr, Hans Jürgen Prömel, and Angelika Steger. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1367. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 29–39. ISBN: 978-3-540-64201-5 978-3-540-69701-5. DOI: 10.1007/BFb0053012. URL: http://link.springer.com/10.1007/BFb0053012 (visited on 07/28/2021).

Arora, Sanjeev and Boaz Barak (2009). *Computational Complexity: A Modern Approach*. Cambridge: Cambridge University Press. ISBN: 978-0-511-80409-0. DOI: 10.1017/CBO9780511804090. URL: http://ebooks.cambridge.org/ref/id/CBO9780511804090 (visited on 04/30/2021).

Besicovitch, A.S. (1935). "On the density of certain sequences of integers". In: *Mathematische Annalen* 110, pp. 336–341. URL: http://eudml.org/doc/159727.

Demillo, Richard A. and Richard J. Lipton (June 1978). "A probabilistic remark on algebraic program testing". In: *Information Processing Letters* 7.4, pp. 193–195. ISSN: 00200190. DOI: 10.1016/0020-0190(78)90067-4. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019078900674 (visited on 05/15/2021).

Erdõs, P. (1935). “Note on the sequences of integers no one of which is divisible by any other”. In: *J. London Math. Soc.* 10, pp. 126–128.

— (1960). “Ob odnom asimptoticheskom neravenstve v teorii tschisel (An asymptotic inequality in the theory of numbers, in Russian)”. In: *Vestnik Leningrad Univ.* 15, 41–49 (Russian).

Etessami, Kousha, Alistair Stewart, and Mihalis Yannakakis (May 2014). “A Note on the Complexity of Comparing Succinctly Represented Integers, with an Application to Maximum Probability Parsing”. In: *ACM Transactions on Computation Theory* 6.2, pp. 1–23. ISSN: 1942-3454, 1942-3462. DOI: 10.1145/2601327. URL: https://dl.acm.org/doi/10.1145/2601327 (visited on 04/30/2021).

Ford, Kevin (Sept. 1, 2008). “The distribution of integers with a divisor in a given interval”. In: *Annals of Mathematics* 168.2, pp. 367–433. ISSN: 0003-486X. DOI: 10.4007/annals.2008.168.367. URL: http://annals.math.princeton.edu/2008/168-2/p01 (visited on 08/13/2021).

Gonnet, Gaston H. (1984). “Determining equivalence of expressions in random polynomial time”. In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing - STOC ’84*. the sixteenth annual ACM symposium. Not Known: ACM Press, pp. 334–341. ISBN: 978-0-89791-133-7. DOI: 10.1145/800057.808698. URL: http://portal.acm.org/citation.cfm?doid=800057.808698 (visited on 05/15/2021).

Kabanets, Valentine and Russell Impagliazzo (Dec. 2004). “Derandomizing Polynomial Identity Tests Means Proving Circuit Lower Bounds”. In: *computational complexity* 13.1, pp. 1–46. ISSN: 1016-3328, 1420-8954. DOI: 10.1007/s00037-004-0182-6. URL: http://link.springer.com/10.1007/s00037-004-0182-6 (visited on 04/30/2021).

Niven, Ivan Morton, Herbert S. Zuckerman, and Hugh L. Montgomery (1991). *An introduction to the theory of numbers*. 5th ed. New York: Wiley. 529 pp. ISBN: 978-0-471-62546-9.

Ramanujan, S. (1915). “Highly Composite Numbers”. In: *Proceedings of the London Mathematical Society* s2_14.1, pp. 347–409. ISSN: 00246115. DOI: 10.1112/plms/s2_14.1.347. URL: http://doi.wiley.com/10.1112/plms/s2_14.1.347 (visited on 08/13/2021).

Regan, Rick (Dec. 2012). “Number of Bits in a Decimal Integer”. In: *Exploring Binary*, p. 11. URL: https://www.exploringbinary.com/number-of-bits-in-a-decimal-integer/ (visited on 08/02/2021).

Schönhage, Arnold (1979). “On The Power of Random Access Machines”. In: *Automata, languages and programming*. H.A. Maurer. ICALP’79 71. LNCS, pp. 520–529.

Schwartz, J. T. (Oct. 1980). “Fast Probabilistic Algorithms for Verification of Polynomial Identities”. In: *Journal of the ACM* 27.4, pp. 701–717. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/322217.322225. URL: https://dl.acm.org/doi/10.1145/322217.322225 (visited on 04/30/2021).

Shpilka, Amir and Amir Yehudayo (n.d.). “Arithmetic Circuits: a survey of recent results and open questions”. In: *Foundations and Trends in Theoretical Computer Science* (), p. 123.

Sloane, N. J. A. (2021). “Sequence A002201/M1591 Superior highly composite numbers”. In: *The On-Line Encyclopedia Of Integer Sequences* (). URL: https://oeis.org/A002201 (visited on 08/18/2021).

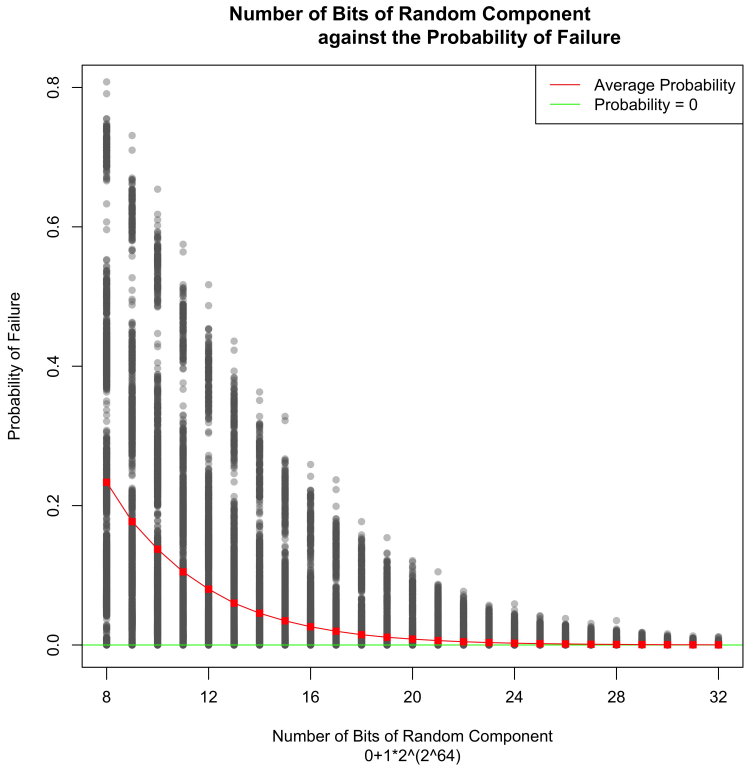
Wigderson, Avi (2019). *Mathematics and computation*. Princeton, NJ: Princeton University Press. ISBN: 978-0-691-18913-0.

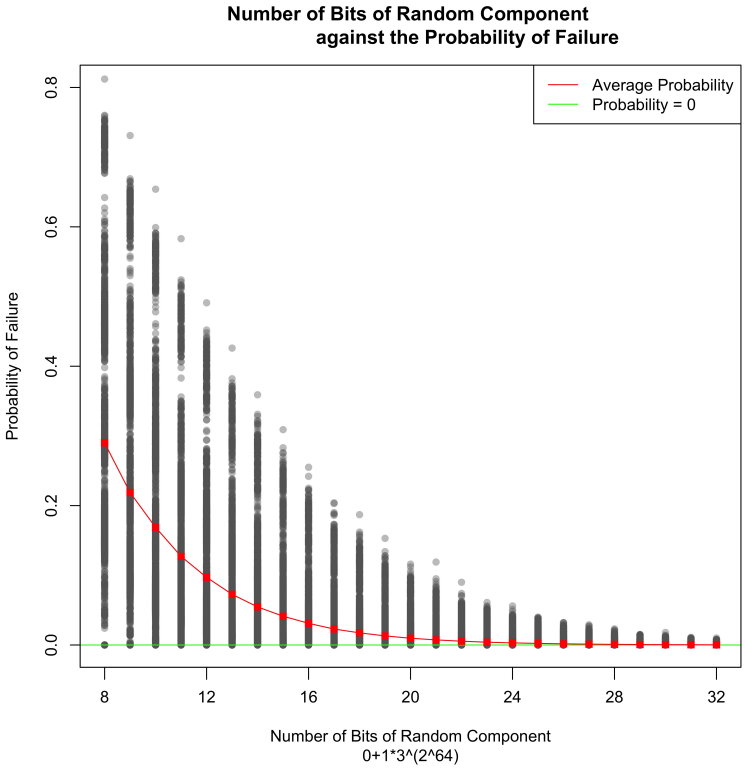
Zippel, Richard (1979). “Probabilistic algorithms for sparse polynomials”. In: *Symbolic and Algebraic Computation*. Ed. by Edward W. Ng. Red. by G. Goos et al. Vol. 72. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 216–226. ISBN: 978-3-540-09519-4 978-3-540-35128-3. DOI: 10.1007/3-540-09519-5_73. URL: http://link.springer.com/10.1007/3-540-09519-5_73 (visited on 04/30/2021).

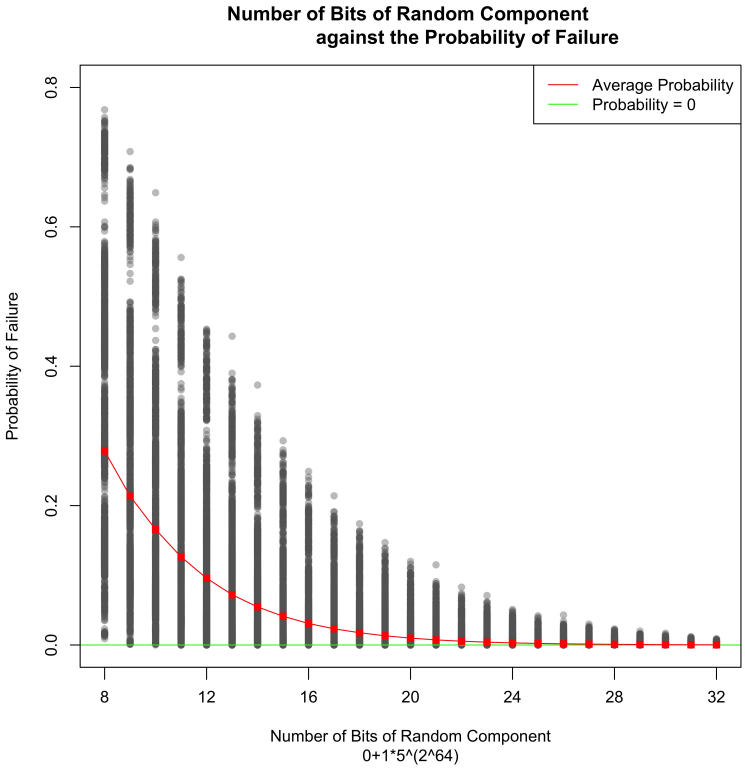
Appendix A

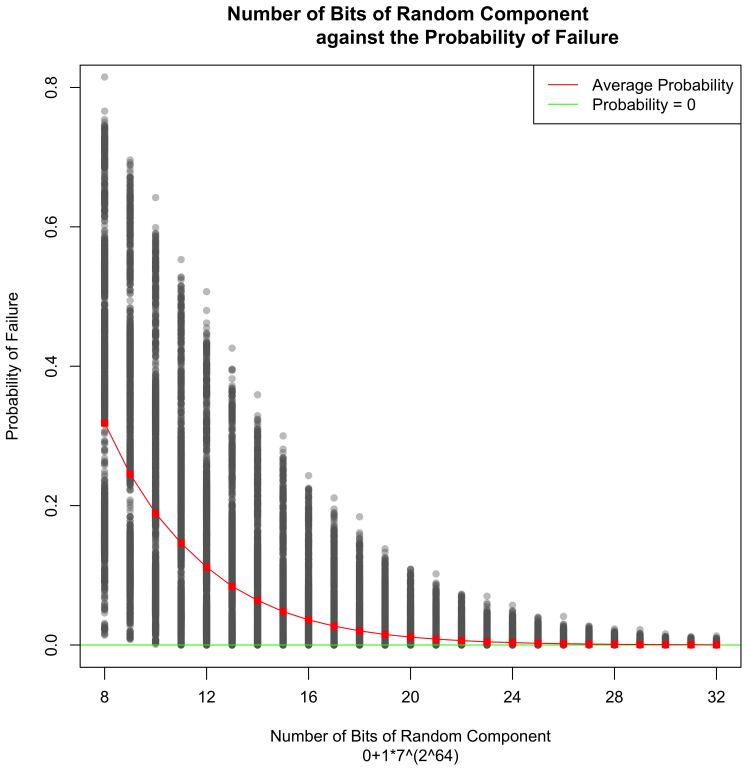
Graphs

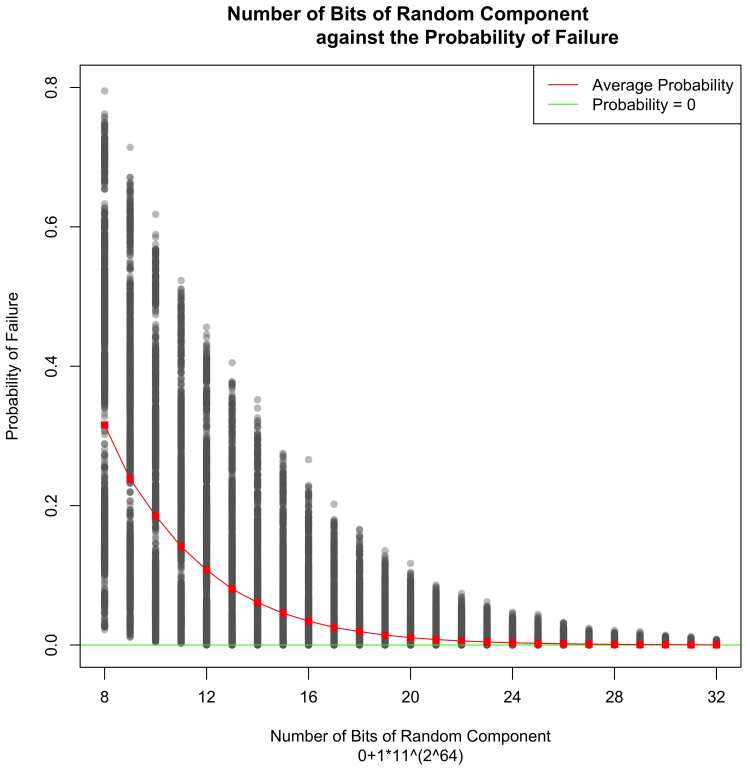
A.1 $p^{2^{1,\ldots,64}} \stackrel{?}{=} p^{2^{1,\ldots,63}}$, for all primes $p < 100$

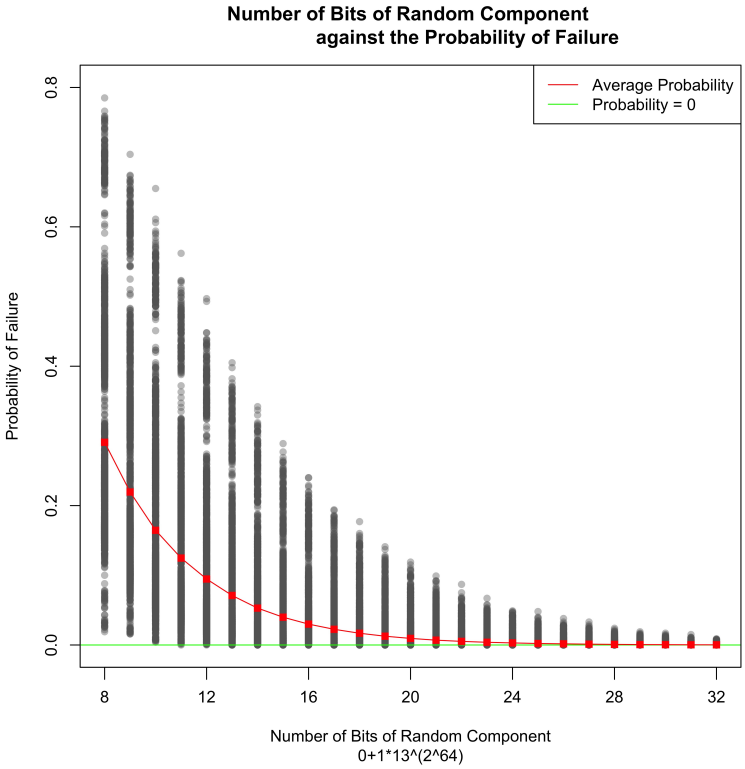


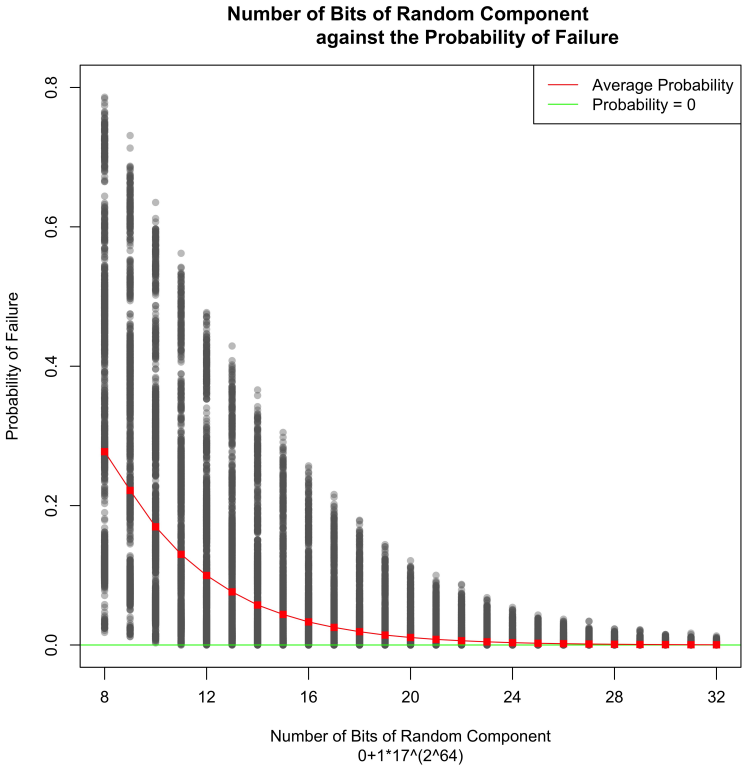

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*3^(2^64)

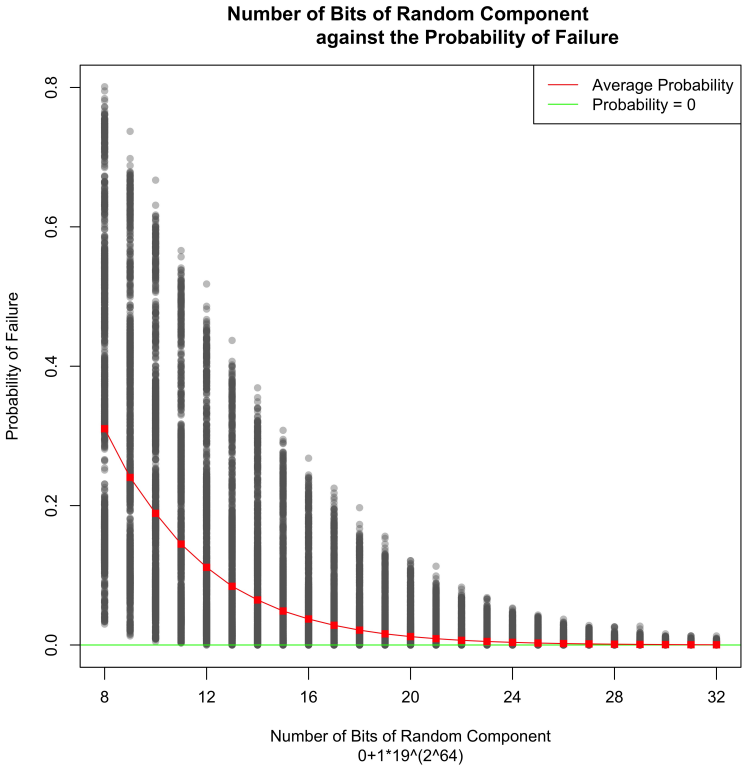

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*5^(2^64)

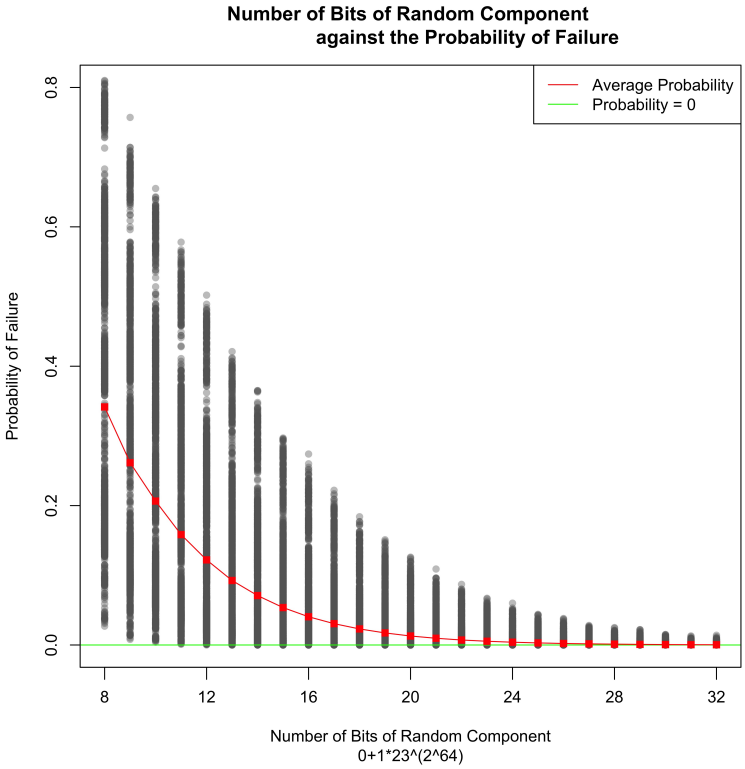

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*7^(2^64)

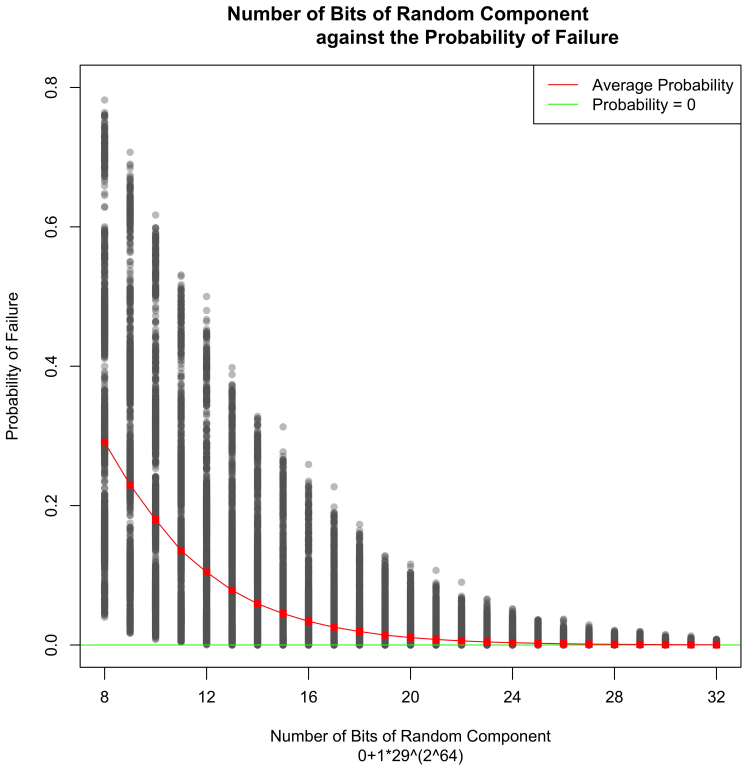

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*11^(2^64)

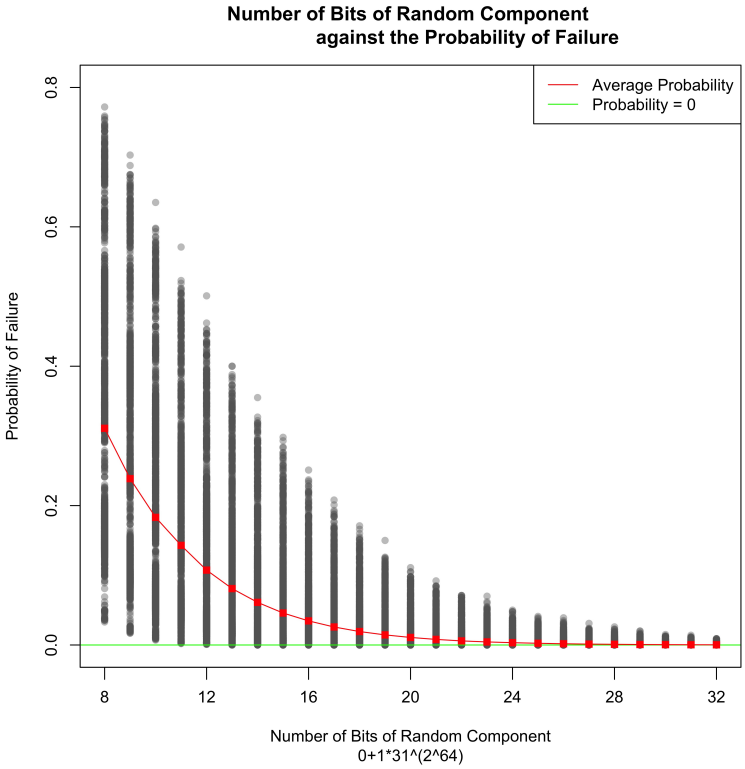

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*13^(2^64)

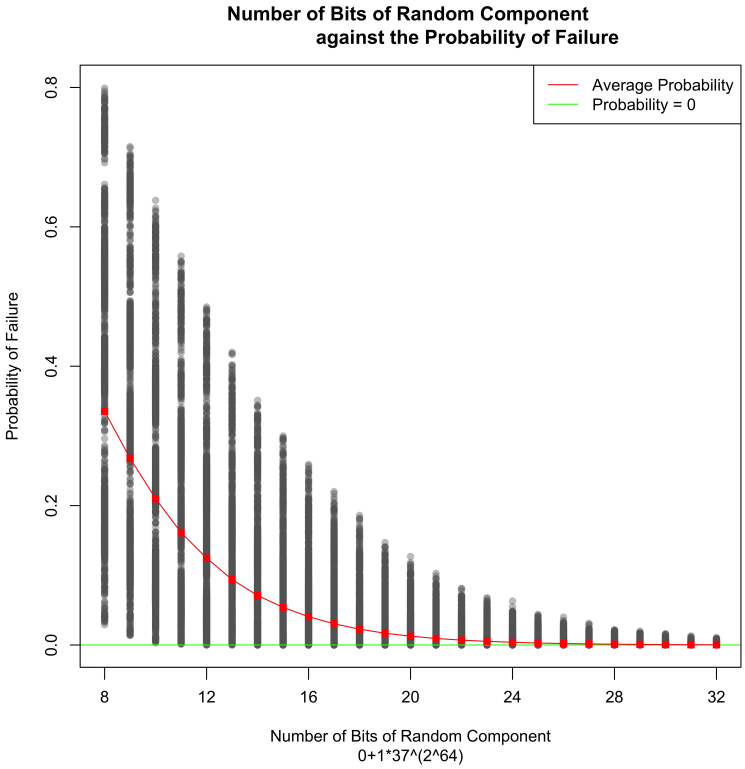

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*17^(2^64)

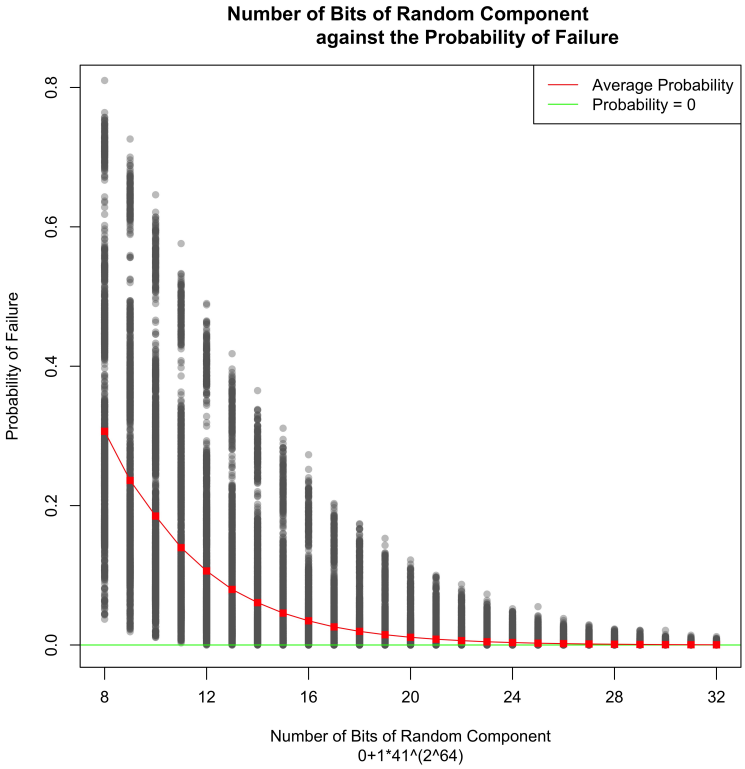

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*19^(2^64)

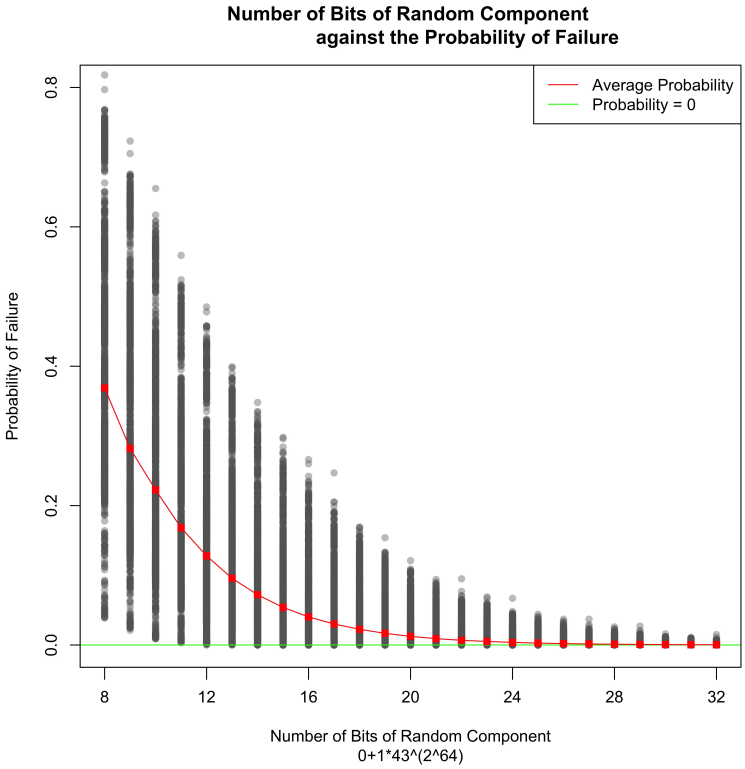

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*23^(2^64)

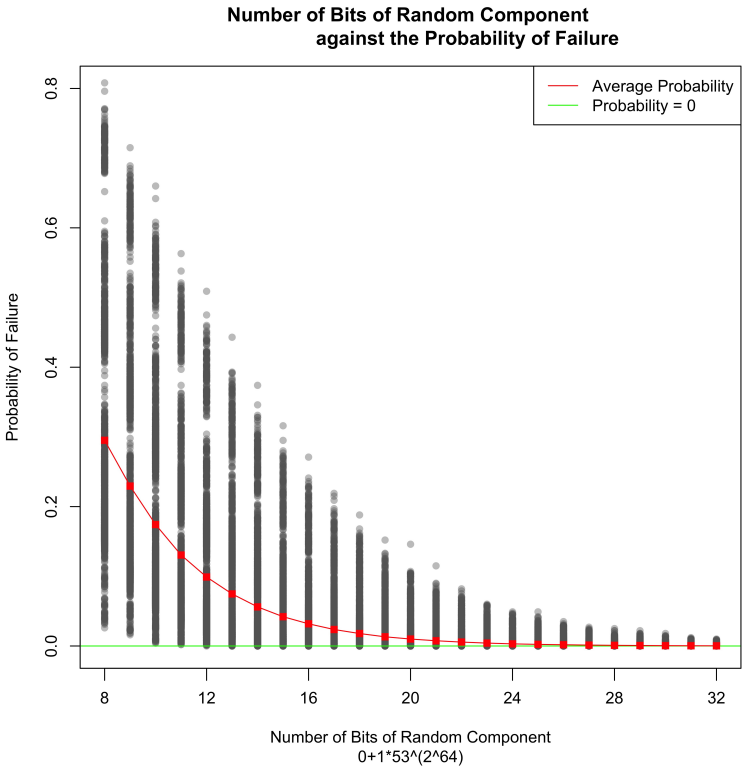

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*29^(2^64)

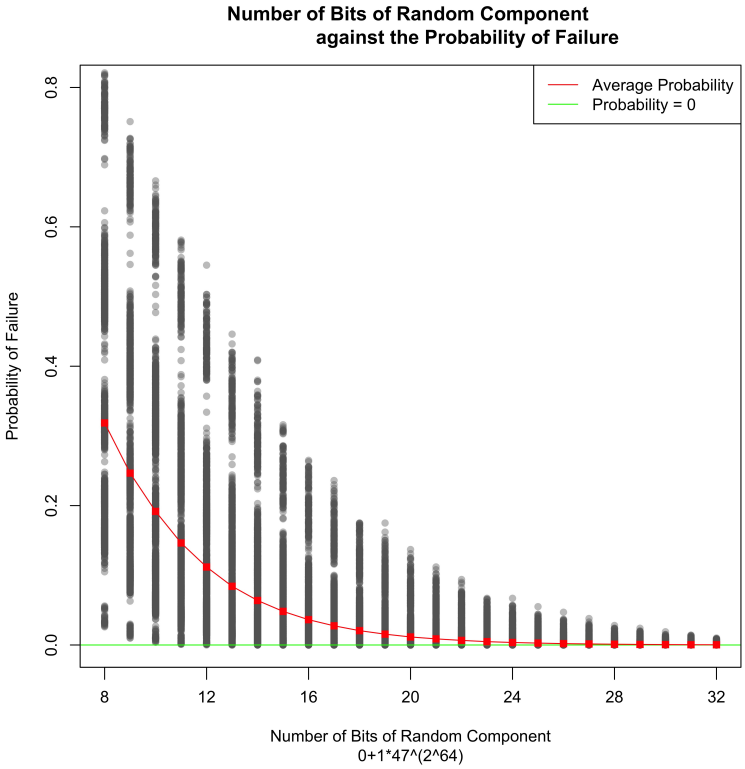

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*31^(2^64)

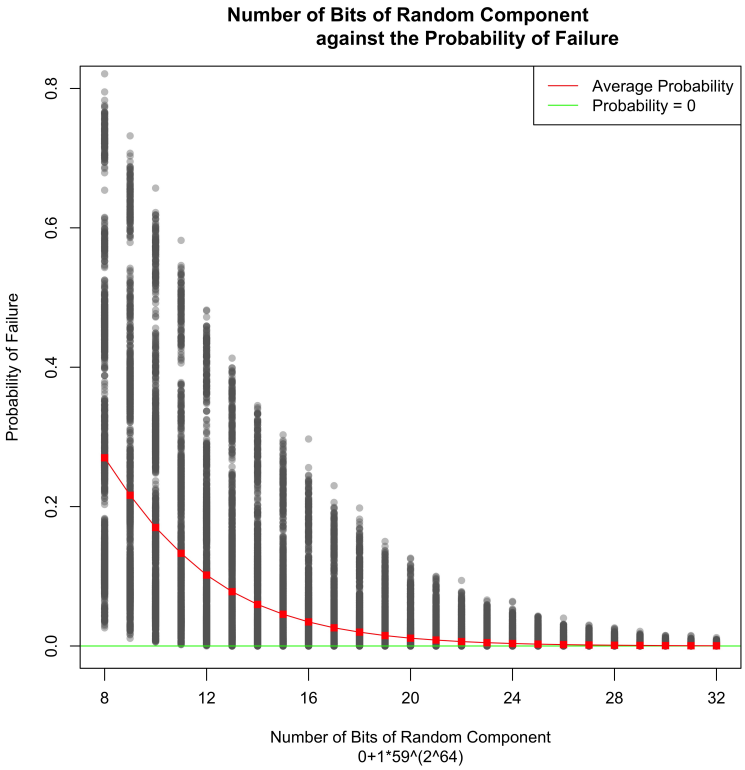

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*37^(2^64)

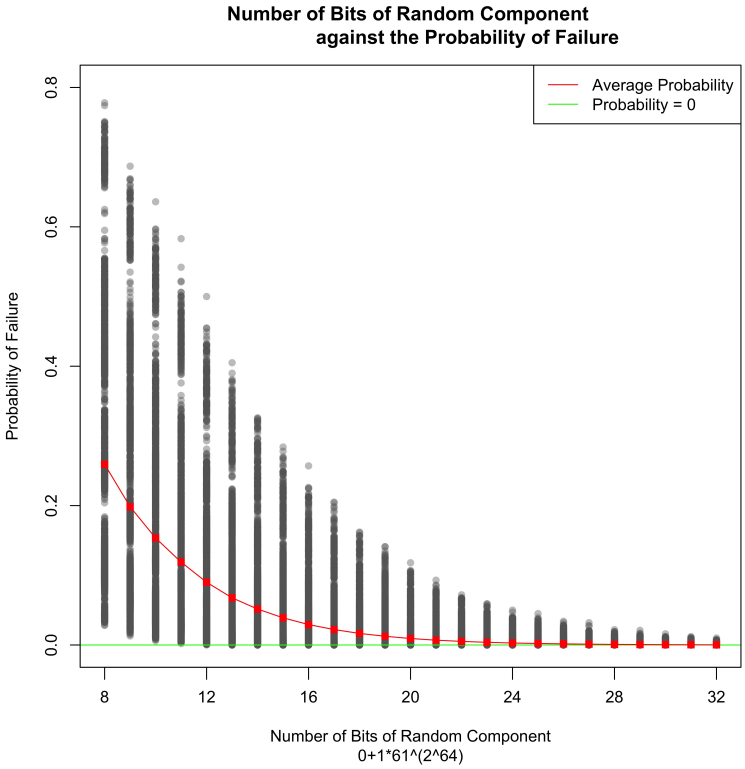

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*41^(2^64)

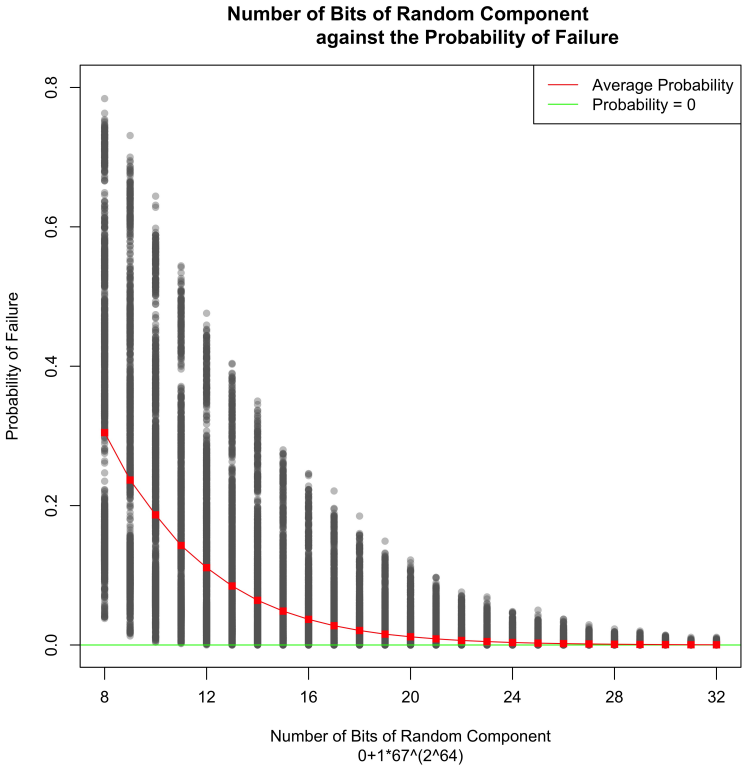

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*43^(2^64)

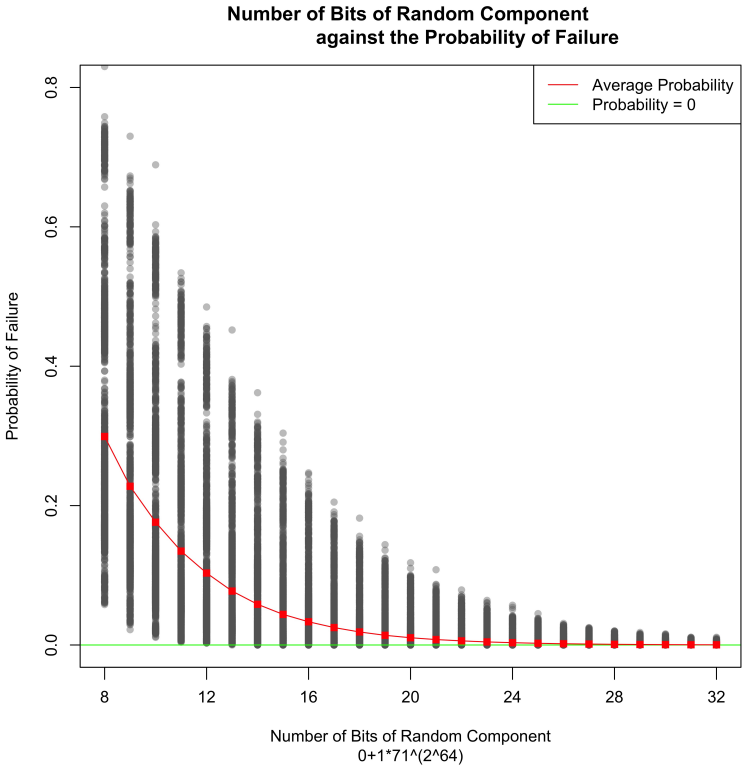

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*53^(2^64)

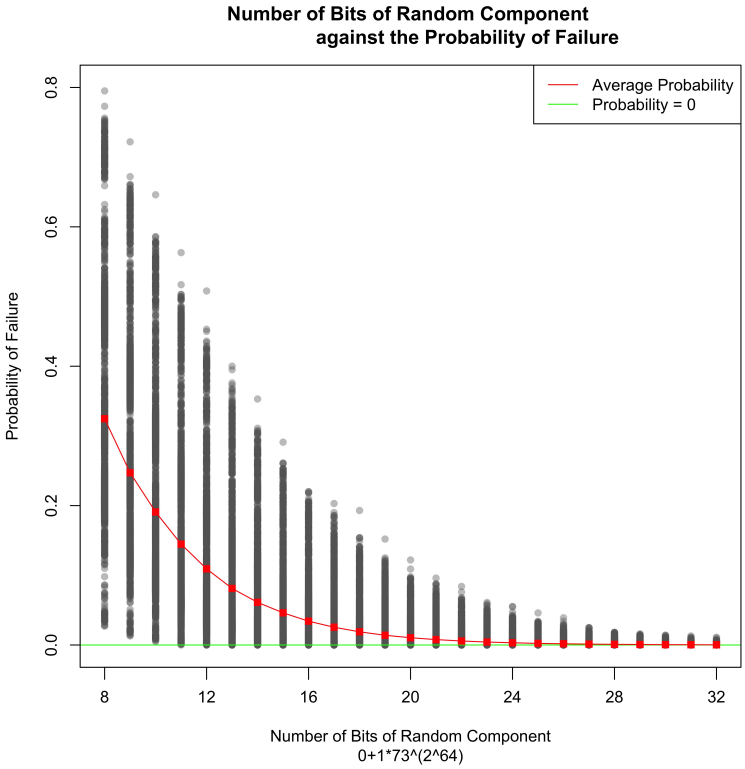

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*47^(2^64)

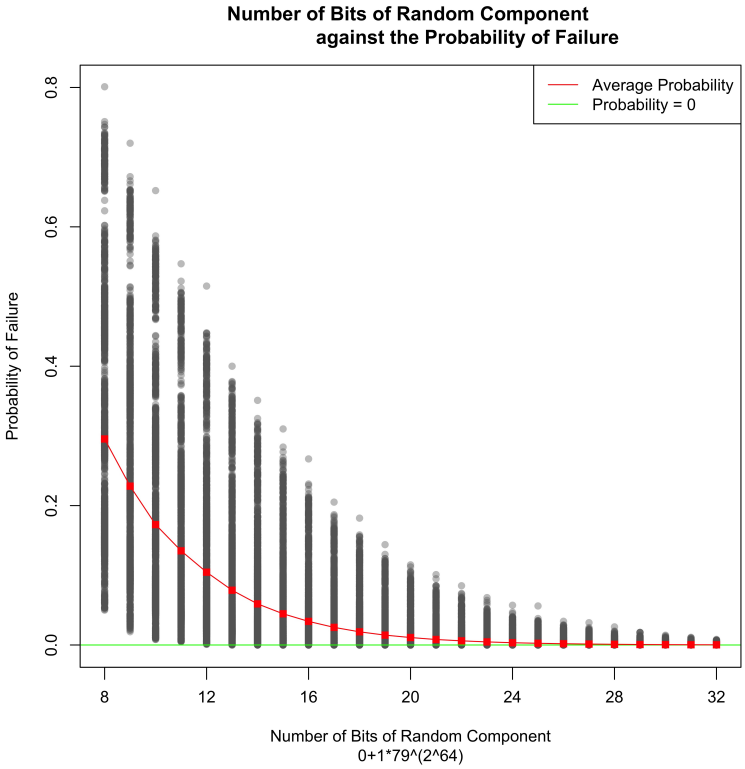

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*59^(2^64)

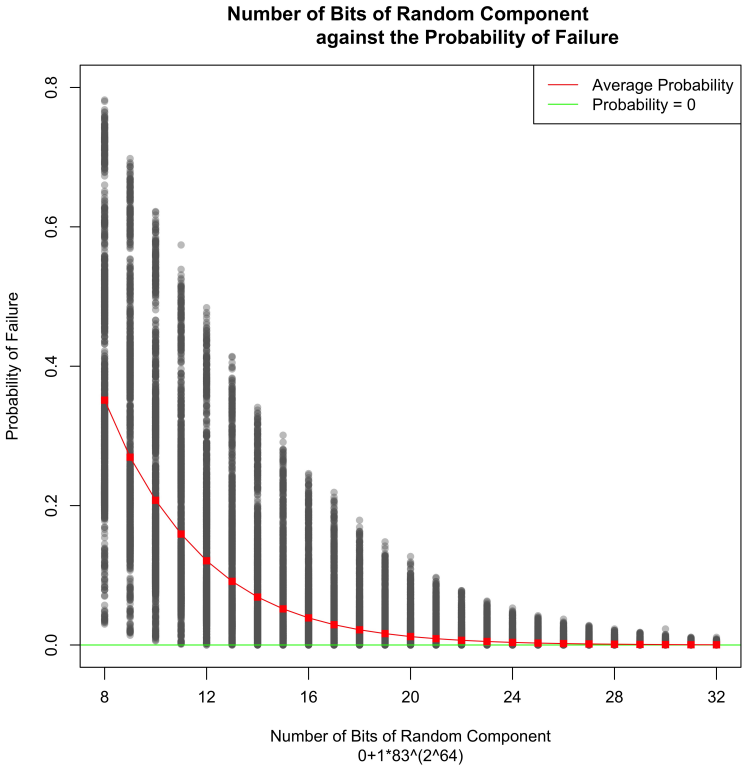

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*61^(2^64)

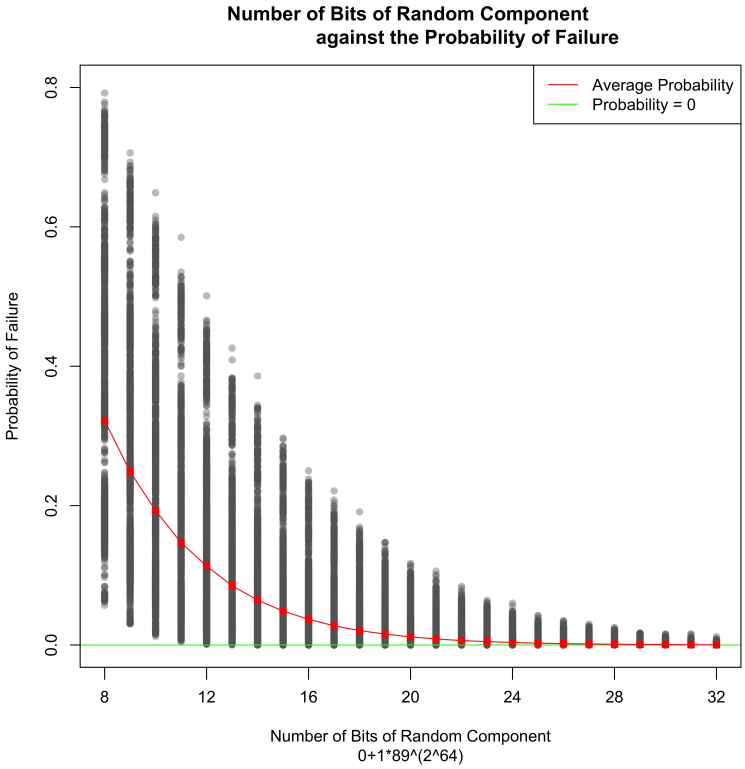

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*67^(2^64)

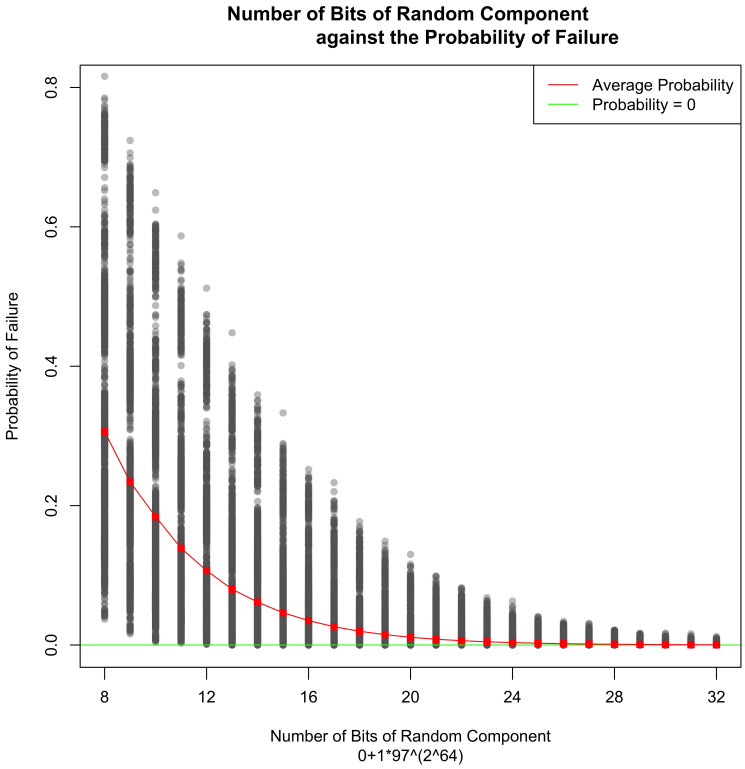

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*71^(2^64)

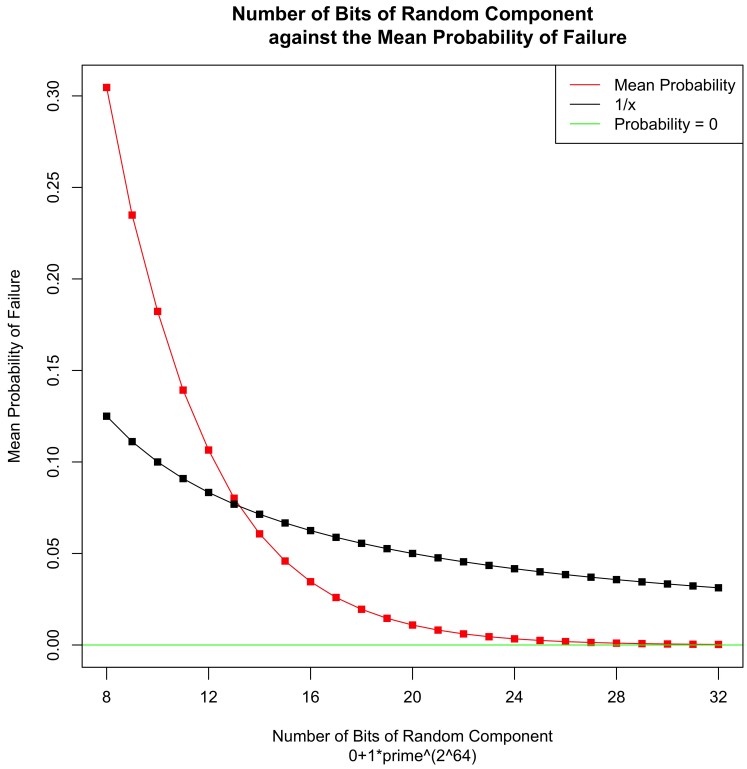

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*73^(2^64)


Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*79^(2^64)

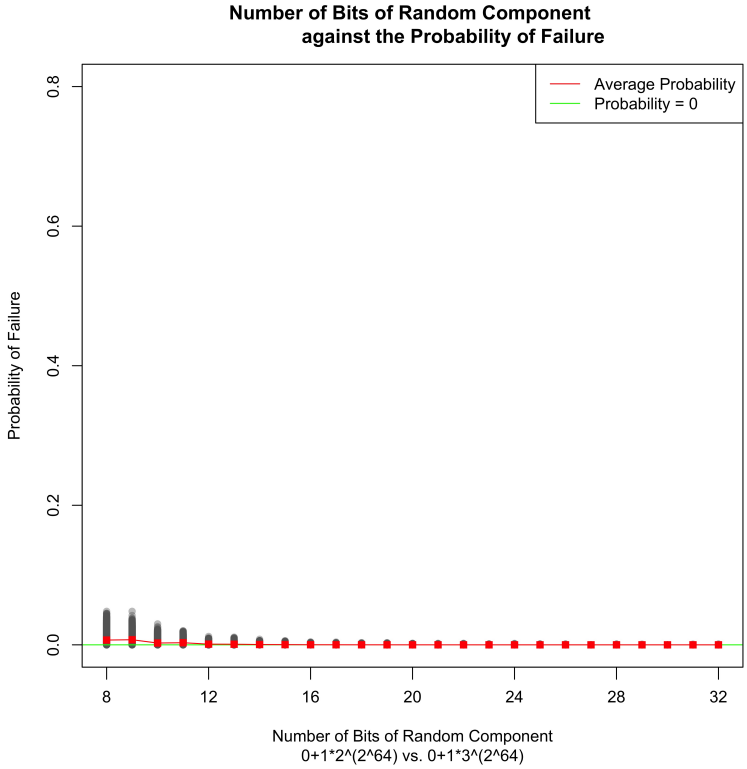

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*83^(2^64)

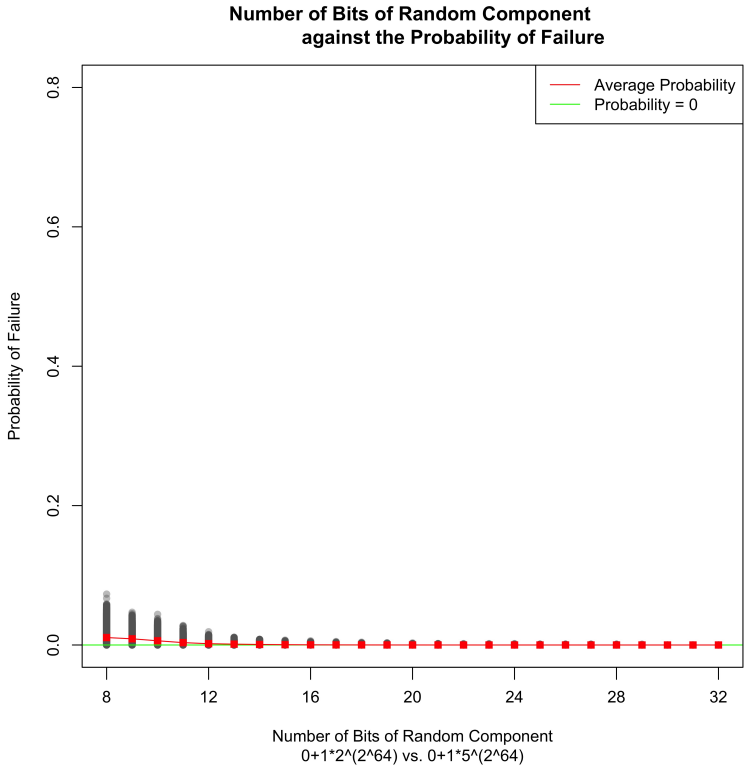

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*89^(2^64)

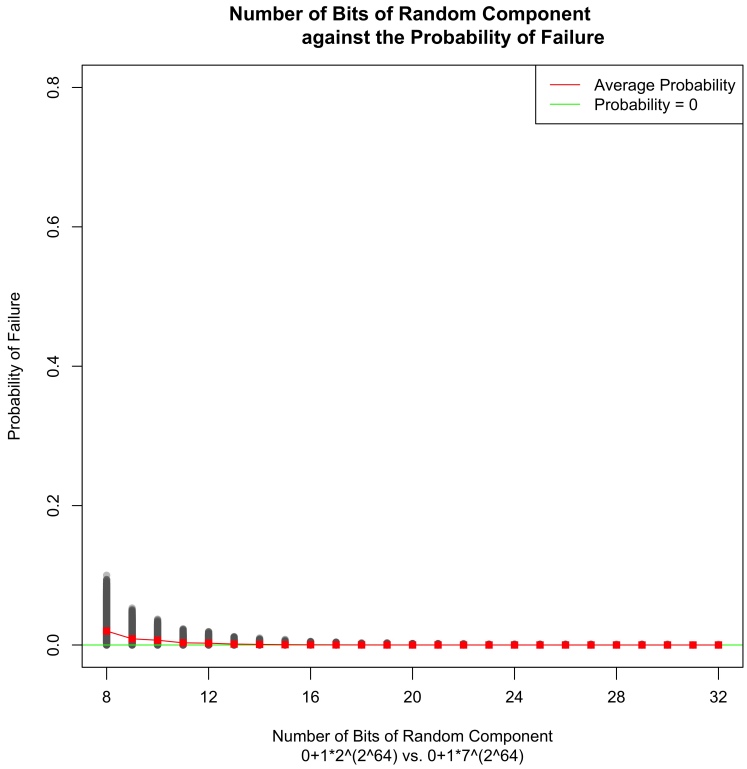

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*97^(2^64)

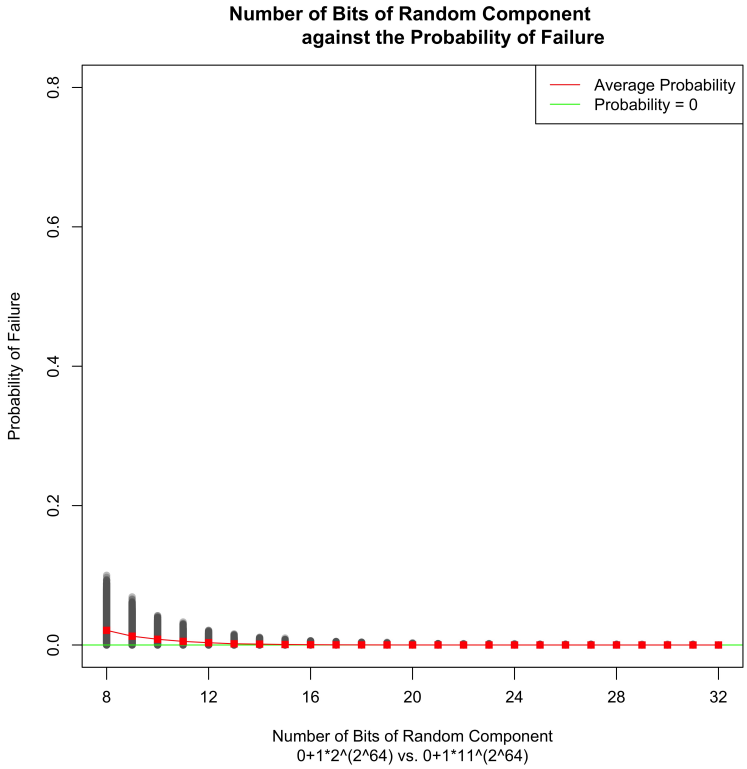


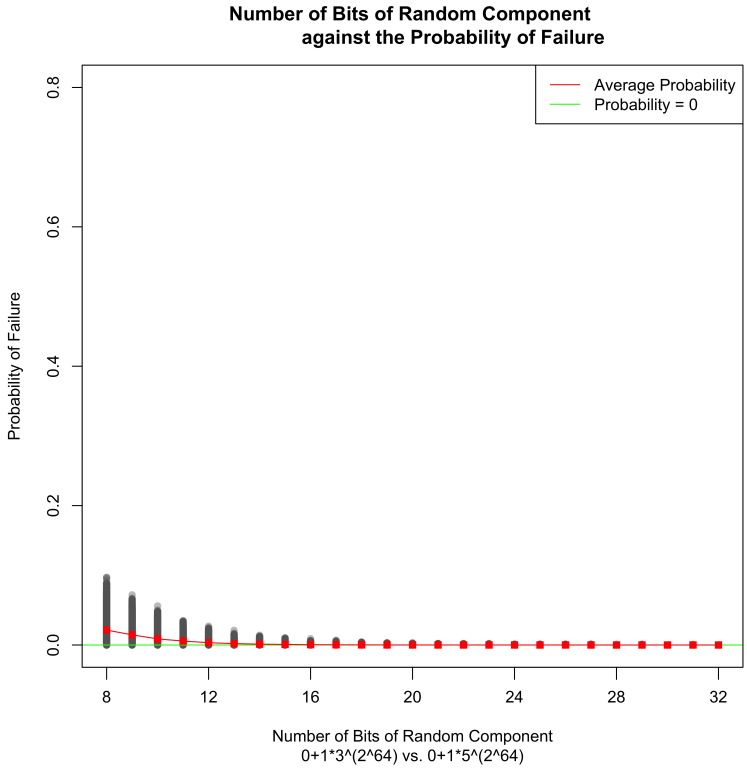
A.2 $p^{2^{1,\ldots,64}} \stackrel{?}{=} q^{2^{1,\ldots,64}}$, for all primes $p, q \leq 11, p \neq q$

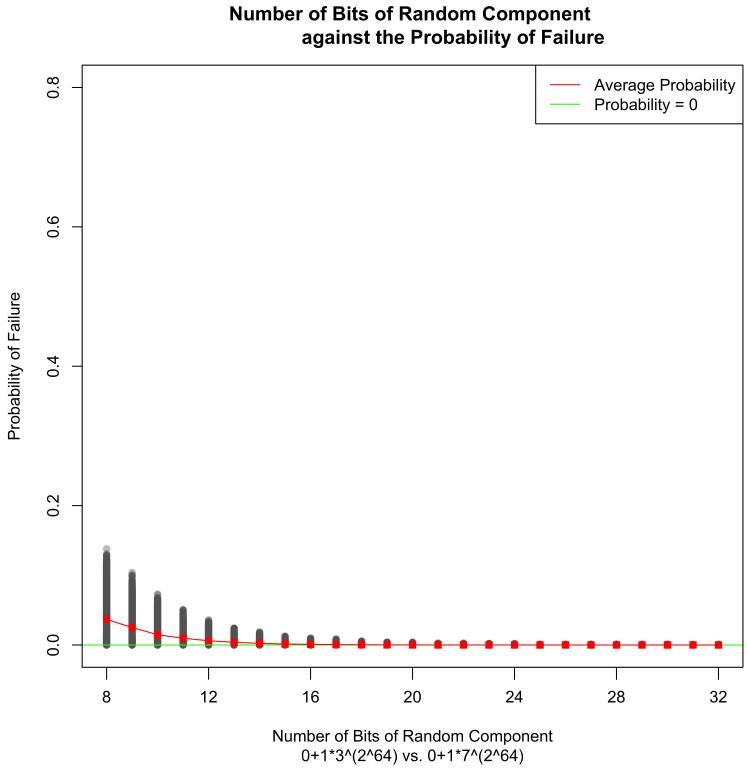


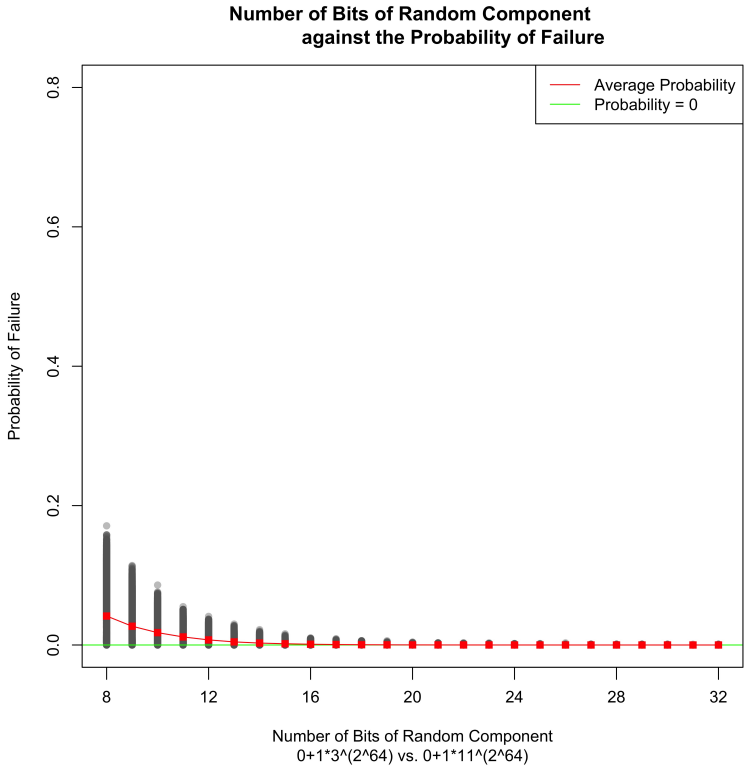

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*2^(2^64) vs. 0+1*5^(2^64)

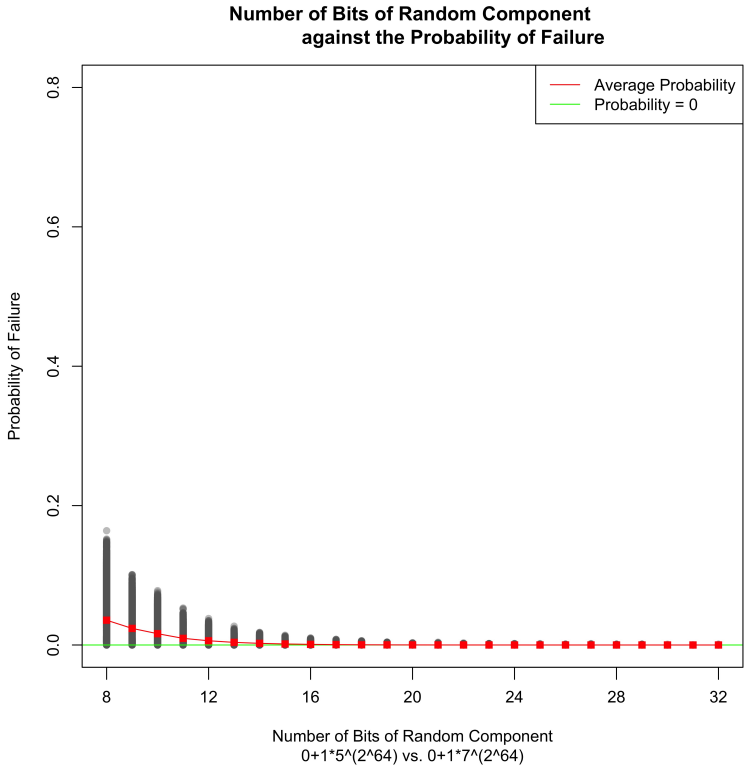

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*2^(2^64) vs. 0+1*7^(2^64)

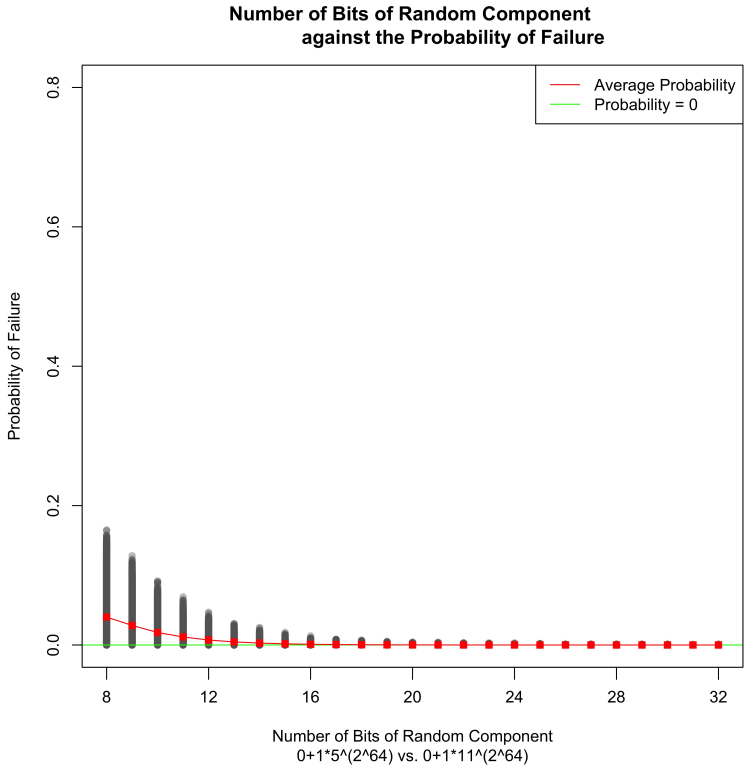

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*2^(2^64) vs. 0+1*11^(2^64)

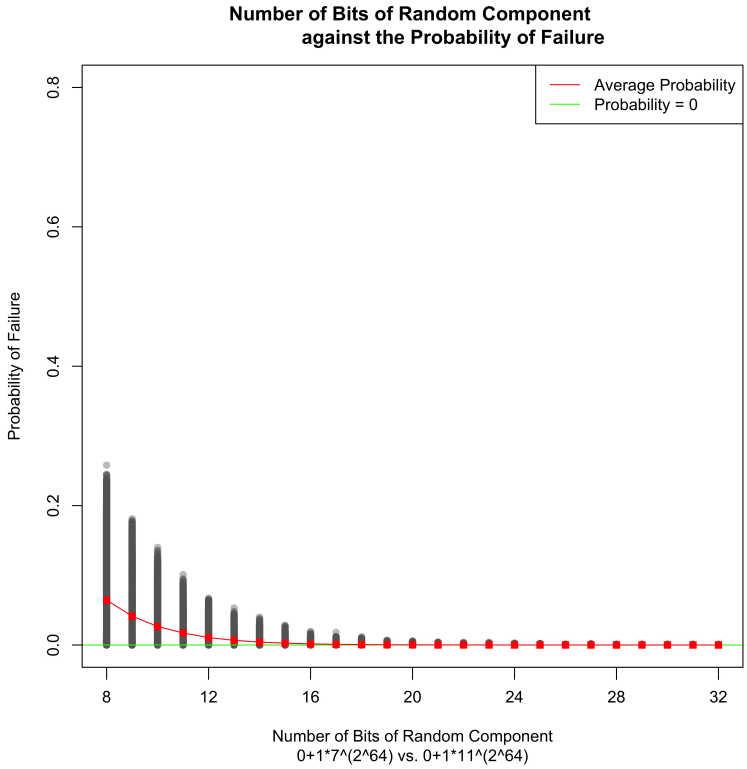

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*3^(2^64) vs. 0+1*5^(2^64)

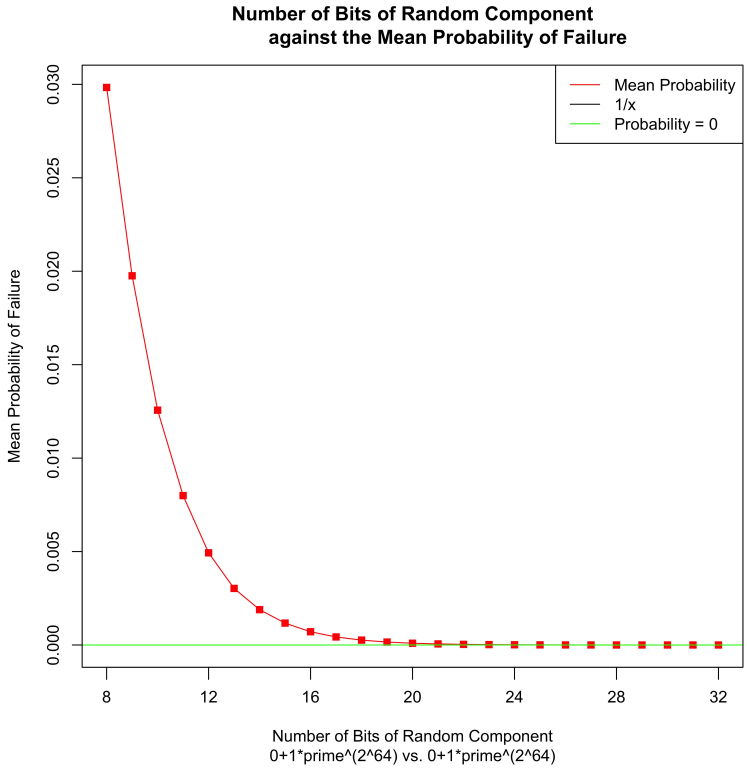

Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*3^(2^64) vs. 0+1*7^(2^64)


Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*3^(2^64) vs. 0+1*11^(2^64)


Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*5^(2^64) vs. 0+1*7^(2^64)


Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*5^(2^64) vs. 0+1*11^(2^64)


Number of Bits of Random Component
against the Probability of Failure
Average Probability
Probability = 0
Probability of Failure
0.0
0.2
0.4
0.6
0.8
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*7^(2^64) vs. 0+1*11^(2^64)


Number of Bits of Random Component
against the Mean Probability of Failure
Mean Probability
1/x
Probability = 0
Mean Probability of Failure
0.000
0.005
0.010
0.015
0.020
0.025
0.030
8
12
16
20
24
28
32
Number of Bits of Random Component
0+1*prime^(2^64) vs. 0+1*prime^(2^64)

Appendix B

Code

B.1 EquSLPData.py

```
#!/usr/bin/env python3

"""
EquSLPData is used to calculate the probability of failure
of the EquSLP algorithm.
"""

import csv
from random import randint

runs = 1000 # number of times to run test in specific range


def EquSLP(X,Y,R):

    """
    EquSLP is used to decide if two Straight Line Programs are
    equal by simply calculating each arithmetic operation of each
    SLP modulo R. If both SLPs have the same modulo R, then they
    might be equal. This process is run many times with different
    R values, if it is always equal, we can conclude that both SLPs
    are equal with high probability and return True; otherwise,
    return False.

    Input: two SLPs X and Y. All SLPs start with 1.
    If X is x0 = 1, x1 = x0 + x0 and x2 = x1 * x1
    then X = [1, (0,0,"+"), (1,1,"*")].
    """
```

```
29
30      # first gate is always 1
31      modX = [1] # list of all the gates in X mod R
32      modY = [1] # list of all the gates in Y mod R
33
34
35      # calculate the mod of each gate in X
36      for i in range(1,len(X)):
37
38          if X[i][2] == "*":
39
40              modX.append( ( modX[X[i][0]] * modX[X[i][1]] ) % R )
41
42          elif X[i][2] == "+":
43
44               modX.append( ( modX[X[i][0]] + modX[X[i][1]] ) % R )
45
46
47      # calculate the mod of each gate in Y
48      for i in range(1,len(Y)):
49
50          if Y[i][2] == "*":
51
52              modY.append( ( modY[Y[i][0]] * modY[Y[i][1]] ) % R )
53
54          elif Y[i][2] == "+":
55
56               modY.append( ( modY[Y[i][0]] + modY[Y[i][1]] ) % R )
57
58
59      # the last gate mod R is equal to the whole SLP mod R
60      if modX[-1] == modY[-1]:
61          return True
62      else:
63          return False
64
65
66
67  def TotalProbability(X,Y,fname):
68
69      """
70       TotalProbability checks every integer in ranges
71       of [2^(i - 1), 2^(i) - 1] up to 2^(2^(n-1)) - 1
72       where n+1 is the full lenght of the SLP.
73       The ranges correspond to all the integers
74       that can be represented using i bits.
```

```
    It calculates the probability of failure of each range.
    """

    # checks in blocks of size [2^(i - 1), 2^(i) - 1]
    for i in range(2, 2**(max(len(X), len(Y)) - 2) + 1):

        S = 0 # total number of integers that give a worng result
        for R in range(2**(i - 1), 2**(i)):

            if EquSLP(X,Y,R):
                S += 1

        probability = S / (2**(i) - 2**(i - 1))

        with open(fname, mode='a') as f:
            f_w = csv.writer(f, delimiter=',', quotechar='"',
                             quoting=csv.QUOTE_MINIMAL)
            f_w.writerow([len(X), len(Y), i, probability])



def RandomProbability(X,Y,fname):

    """
    RandomProbability pseudorandomly selects "runs"
    amount of integers from ranges [2^(i - 1), 2^(i) - 1]
    up to 2^(2^(n-1)) - 1 where n+1 is the full lenght of
    the SLP. The ranges correspond to all the integers
    that can be represented using i bits.
    It calculates the probability of failure of each range.
    """

    # checks in blocks of size [2^(i - 1), 2^(i) - 1]
    for i in range(2, 2**(max(len(X), len(Y)) - 2) + 1):

        S = 0 # total number of integers that give a worng result
        for _ in range(runs):

            # generate random integer
            R = randint(2**(i - 1), 2**(i) - 1)

            if EquSLP(X,Y,R):
                S += 1

        with open(fname, mode='a') as f:
            f_w = csv.writer(f, delimiter=',', quotechar='"',
```

```
                                quoting=csv.QUOTE_MINIMAL)
            f_w.writerow([len(X), len(Y), i, S / runs ])


def TotalProbabilityLimited(X,Y,fname,u,v):

    """
    TotalProbabilityLimited checks every integer in ranges
    of [2^(i - 1), 2^(i) - 1] where i is in [u-1,v].
    The ranges correspond to all the integers
    that can be represented using i bits.
    It calculates the probability of failure of each range.
    """

    # checks in blocks of size [2^(i - 1), 2^(i) - 1]
    for i in range(u, v + 1):

        S = 0 # total number of integers that give a worng result
        for R in range(2**(i - 1), 2**(i)):

            if EquSLP(X,Y,R):
                S += 1

        probability = S / (2**(i) - 2**(i - 1))

        with open(fname, mode='a') as f:
            f_w = csv.writer(f, delimiter=',', quotechar='"',
                             quoting=csv.QUOTE_MINIMAL)
            f_w.writerow([len(X), len(Y), i, probability])


def RandomProbabilityLimited(X,Y,fname,u,v):

    """
    RandomProbabilityLimited pseudorandomly selects "runs"
    amount of integers from ranges [2^(i - 1), 2^(i) - 1]
    where i is in [u-1,v].
    The ranges correspond to all the integers
    that can be represented using i bits.
    It calculates the probability of failure of each range.
    """

    # checks in blocks of size [2^(i - 1), 2^(i) - 1]
    for i in range(u, v + 1):
```

```
167
168         S = 0 # total number of integers that give a worng result
169         for _ in range(runs):
170
171             # generate random integer
172             R = randint(2**(i - 1), 2**(i) - 1)
173
174             if EquSLP(X,Y,R):
175                 S += 1
176
177         with open(fname, mode='a') as f:
178             f_w = csv.writer(f, delimiter=',', quotechar='"',
179                              quoting=csv.QUOTE_MINIMAL)
180             f_w.writerow([len(X), len(Y), i, S / runs ])
```

B.2 Test.py

```
1  #!/usr/bin/env python3
2
3  """
4  Test is made of many functions that represent different tests.
5  """
6
7  from EquSLPData import *
8  import csv
9  import math
10
11
12 def TotalValue(X):
13
14     """
15      Calculates the explicit value of an SLP X
16      """
17
18     # first gate is always 1
19     totalX = [1]
20
21     for i in range(1,len(X)):
22
23         if X[i][2] == "*":
24
```

```
            totalX.append((totalX[X[i][0]] * totalX[X[i][1]]))

        elif X[i][2] == "+":

            totalX.append((totalX[X[i][0]] + totalX[X[i][1]]))

    return totalX[-1]



def TestPremiumA(a,b,c):

    """
    Applies EquSLPData to:
    - a+b*c^(2^n) vs. { a+b*c^(2^(n-1)) ; a+b*c^(2^(n-2))
                        ; ... ; a+b*c^(2^1) }
    - a+b*c^(2^(n-1)) vs. { a+b*c^(2^(n-2)) ; a+b*c^(2^(n-3))
                            ; ... ; a+b*c^(2^1) }
    - ...
    - a+b*c^(2^2) vs. a+b*c^(2^1)
    """

    # iterations = (n^2 - n) / 2
    n = 64
    # [2^(i - 1), 2^(i) - 1] where i is in [u-1,v]
    u = 8
    v = 32

    if n < 1 or a < 0 or b < 1 or c < 1 or u < 1 or v < u:
        return False

    fname = str(a)+"+"+str(b)+"*"+str(c)+"^(2^"+str(n)+").csv"

    with open(fname, mode='w') as f:
        f_w = csv.writer(f, delimiter=',',
                         quotechar='"', quoting=csv.QUOTE_MINIMAL)
        f_w.writerow(['lenX', 'lenY', "bits", "proba"])


    X = [1]
    for i in range(max(a,b,c)-1):

        X.append((i,0,'+'))

    for i in range(n):

```

```
        X.append((c-1+i,c-1+i,'*'))


    if a == 0 and b == 1:

        for _ in range(n):

            Y = X
            for _ in range(n):

                if len(X) != len(Y):
                    RandomProbabilityLimited(X,Y,fname,u,v)

                if len(Y)-1 == max(a,b,c):
                    break

                Y = Y[:len(Y)-1]

            X = X[:len(X)-1]


    elif a == 0 and b > 1:

        for _ in range(n):

            X.append((X[-1][0]+1,b-1,'*'))

            Y = X
            for _ in range(n):

                if len(X) != len(Y):
                    RandomProbabilityLimited(X,Y,fname,u,v)

                if len(Y)-2 == max(a,b,c):
                    break

                Y = Y[:len(Y)-2]
                Y.append((Y[-1][0]+1,b-1,'*'))

            X = X[:len(X)-2]


    else:

        for _ in range(n):

```

```
            X.append((X[-1][0]+1,b-1,'*'))
            X.append((X[-1][0]+1,a-1,'+'))

            Y = X
            for _ in range(n):

                if len(X) != len(Y):
                    RandomProbabilityLimited(X,Y,fname,u,v)

                if len(Y)-3 == max(a,b,c):
                    break

                Y = Y[:len(Y)-3]
                Y.append((Y[-1][0]+1,b-1,'*'))
                Y.append((Y[-1][0]+1,a-1,'+'))

            X = X[:len(X)-3]



def TestPremiumB(a,b,c,x,y,z):

    """
    Applies EquSLPData to:
    - a+b*c^(2^n) vs. { x+y*z^(2^n) ; x+y*z^(2^(n-1))
                        ; ... ; x+y*z^(2^(1)) }
    - a+b*c^(2^(n-1) vs. { x+y*z^(2^n) ; x+y*z^(2^(n-1))
                           ; ... ; x+y*z^(2^(1)) }
    - ...
    - a+b*c^(2^1) vs. { x+y*z^(2^n) ; x+y*z^(2^(n-1))
                        ; ... ; x+y*z^(2^(1)) }
    """

    # iterations = n^2
    n = 64
    # [2^(i - 1), 2^(i) - 1] where i is in [u-1,v]
    u = 8
    v = 32

    if (n < 1 or a < 0 or b < 1 or c < 1 or x < 0 or y < 1 or
        z < 1 or u < 1 or v < u or (a == x and b == y and c == z)):
        return False

    fname = str(a)+"+"+str(b)+"*"+str(c)+"^(2^"+str(n)+") vs. "
    +str(x)+"+"+str(y)+"*"+str(z)+"^(2^"+str(n)+").csv"

```

```
    with open(fname, mode='w') as f:
        f_w = csv.writer(f, delimiter=',',
                         quotechar='"', quoting=csv.QUOTE_MINIMAL)
        f_w.writerow(['lenX', 'lenY', "bits", "proba"])


    X = [1]
    for i in range(max(a,b,c)-1):

        X.append((i,0,'+'))

    Y = [1]
    for i in range(max(x,y,z)-1):

        Y.append((i,0,'+'))


    if a == 0 and b == 1:


        if x == 0 and y == 1:

            for i in range(n):

                X.append((c-1+i,c-1+i,'*'))

                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
                    RandomProbabilityLimited(X,Y,fname,u,v)

                Y = [1]
                for k in range(max(x,y,z)-1):

                    Y.append((k,0,'+'))


        elif x == 0 and y > 1:

            for i in range(n):

                X.append((c-1+i,c-1+i,'*'))

                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
```

```
                    Y.append((Y[-1][0]+1,y-1,'*'))
                    RandomProbabilityLimited(X,Y,fname,u,v)
                    Y.pop()

                Y = [1]
                for k in range(max(x,y,z)-1):

                    Y.append((k,0,'+'))


        elif x > 0 and y == 1:

            for i in range(n):

                X.append((c-1+i,c-1+i,'*'))

                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
                    Y.append((Y[-1][0]+1,x-1,'+'))
                    RandomProbabilityLimited(X,Y,fname,u,v)
                    Y.pop()

                Y = [1]
                for k in range(max(x,y,z)-1):

                    Y.append((k,0,'+'))


        else:

               for i in range(n):

                X.append((c-1+i,c-1+i,'*'))

                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
                    Y.append((Y[-1][0]+1,y-1,'*'))
                    Y.append((Y[-1][0]+1,x-1,'+'))
                    RandomProbabilityLimited(X,Y,fname,u,v)
                    Y = Y[:len(Y)-2]

                Y = [1]
                for k in range(max(x,y,z)-1):

```

```
                    Y.append((k,0,'+'))



    if a == 0 and b > 1:


        if x == 0 and y == 1:

            for i in range(n):

                X.append((c-1+i,c-1+i,'*'))
                X.append((X[-1][0]+1,b-1,'*'))

                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
                    RandomProbabilityLimited(X,Y,fname,u,v)

                X.pop()
                Y = [1]
                for k in range(max(x,y,z)-1):

                    Y.append((k,0,'+'))


        elif x == 0 and y > 1:

            for i in range(n):

                X.append((c-1+i,c-1+i,'*'))
                X.append((X[-1][0]+1,b-1,'*'))

                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
                    Y.append((Y[-1][0]+1,y-1,'*'))
                    RandomProbabilityLimited(X,Y,fname,u,v)
                    Y.pop()

                X.pop()
                Y = [1]
                for k in range(max(x,y,z)-1):

                    Y.append((k,0,'+'))
```

```
301
302         elif x > 0 and y == 1:
303
304             for i in range(n):
305
306                 X.append((c-1+i,c-1+i,'*'))
307                 X.append((X[-1][0]+1,b-1,'*'))
308
309                 for j in range(n):
310
311                     Y.append((z-1+j,z-1+j,'*'))
312                     Y.append((Y[-1][0]+1,x-1,'+'))
313                     RandomProbabilityLimited(X,Y,fname,u,v)
314                     Y.pop()
315
316                 X.pop()
317                 Y = [1]
318                 for k in range(max(x,y,z)-1):
319
320                     Y.append((k,0,'+'))
321
322
323         else:
324
325                for i in range(n):
326
327                 X.append((c-1+i,c-1+i,'*'))
328                 X.append((X[-1][0]+1,b-1,'*'))
329
330                 for j in range(n):
331
332                     Y.append((z-1+j,z-1+j,'*'))
333                     Y.append((Y[-1][0]+1,y-1,'*'))
334                     Y.append((Y[-1][0]+1,x-1,'+'))
335                     RandomProbabilityLimited(X,Y,fname,u,v)
336                     Y = Y[:len(Y)-2]
337
338                 X.pop()
339                 Y = [1]
340                 for k in range(max(x,y,z)-1):
341
342                     Y.append((k,0,'+'))
343
344
345
346     if a > 0 and b == 1:
```

```
347
348
349         if x == 0 and y == 1:
350
351             for i in range(n):
352
353                 X.append((c-1+i,c-1+i,'*'))
354                 X.append((X[-1][0]+1,a-1,'+'))
355
356                 for j in range(n):
357
358                     Y.append((z-1+j,z-1+j,'*'))
359                     RandomProbabilityLimited(X,Y,fname,u,v)
360
361                 X.pop()
362                 Y = [1]
363                 for k in range(max(x,y,z)-1):
364
365                     Y.append((k,0,'+'))
366
367
368         elif x == 0 and y > 1:
369
370             for i in range(n):
371
372                 X.append((c-1+i,c-1+i,'*'))
373                 X.append((X[-1][0]+1,a-1,'+'))
374
375                 for j in range(n):
376
377                     Y.append((z-1+j,z-1+j,'*'))
378                     Y.append((Y[-1][0]+1,y-1,'*'))
379                     RandomProbabilityLimited(X,Y,fname,u,v)
380                     Y.pop()
381
382                 X.pop()
383                 Y = [1]
384                 for k in range(max(x,y,z)-1):
385
386                     Y.append((k,0,'+'))
387
388
389         elif x > 0 and y == 1:
390
391             for i in range(n):
392
```

```
                X.append((c-1+i,c-1+i,'*'))
                X.append((X[-1][0]+1,a-1,'+'))

                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
                    Y.append((Y[-1][0]+1,x-1,'+'))
                    RandomProbabilityLimited(X,Y,fname,u,v)
                    Y.pop()

                X.pop()
                Y = [1]
                for k in range(max(x,y,z)-1):

                    Y.append((k,0,'+'))


        else:

               for i in range(n):

                X.append((c-1+i,c-1+i,'*'))
                X.append((X[-1][0]+1,a-1,'+'))

                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
                    Y.append((Y[-1][0]+1,y-1,'*'))
                    Y.append((Y[-1][0]+1,x-1,'+'))
                    RandomProbabilityLimited(X,Y,fname,u,v)
                    Y = Y[:len(Y)-2]

                X.pop()
                Y = [1]
                for k in range(max(x,y,z)-1):

                    Y.append((k,0,'+'))



    else:

        if x == 0 and y == 1:

            for i in range(n):

```

```
                X.append((c-1+i,c-1+i,'*'))
                X.append((X[-1][0]+1,b-1,'*'))
                X.append((X[-1][0]+1,a-1,'+'))

                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
                    RandomProbabilityLimited(X,Y,fname,u,v)

                X = X[:len(X)-2]
                Y = [1]
                for k in range(max(x,y,z)-1):

                    Y.append((k,0,'+'))


        elif x == 0 and y > 1:

            for i in range(n):

                X.append((c-1+i,c-1+i,'*'))
                X.append((X[-1][0]+1,b-1,'*'))
                X.append((X[-1][0]+1,a-1,'+'))

                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
                    Y.append((Y[-1][0]+1,y-1,'*'))
                    RandomProbabilityLimited(X,Y,fname,u,v)
                    Y.pop()

                X = X[:len(X)-2]
                Y = [1]
                for k in range(max(x,y,z)-1):

                    Y.append((k,0,'+'))


        elif x > 0 and y == 1:

            for i in range(n):

                X.append((c-1+i,c-1+i,'*'))
                X.append((X[-1][0]+1,b-1,'*'))
                X.append((X[-1][0]+1,a-1,'+'))
```

```
                for j in range(n):

                    Y.append((z-1+j,z-1+j,'*'))
                    Y.append((Y[-1][0]+1,x-1,'+'))
                    RandomProbabilityLimited(X,Y,fname,u,v)
                    Y.pop()

                X = X[:len(X)-2]
                Y = [1]
                for k in range(max(x,y,z)-1):

                    Y.append((k,0,'+'))


        else:

                for i in range(n):

                 X.append((c-1+i,c-1+i,'*'))
                 X.append((X[-1][0]+1,b-1,'*'))
                 X.append((X[-1][0]+1,a-1,'+'))

                 for j in range(n):

                     Y.append((z-1+j,z-1+j,'*'))
                     Y.append((Y[-1][0]+1,y-1,'*'))
                     Y.append((Y[-1][0]+1,x-1,'+'))
                     RandomProbabilityLimited(X,Y,fname,u,v)
                     Y = Y[:len(Y)-2]

                 X = X[:len(X)-2]
                 Y = [1]
                 for k in range(max(x,y,z)-1):

                     Y.append((k,0,'+'))
```

B.3 Divisors.py

```
#!/usr/bin/env python3

def divisors(n):
```

```
4
5       """
6       Author: Tomas Kulich
7       url: https://stackoverflow.com/questions/171765/
8           what-is-the-best-way-to-get-all-the-divisors-of-a-number
9       This function finds all the divisors of an integer
10      """
11
12      # get factors and their counts
13      factors = {}
14      nn = n
15      i = 2
16      while i*i <= nn:
17          while nn % i == 0:
18              factors[i] = factors.get(i, 0) + 1
19              nn //= i
20          i += 1
21      if nn > 1:
22          factors[nn] = factors.get(nn, 0) + 1
23
24      primes = list(factors.keys())
25
26      # generates factors from primes[k:] subset
27      def generate(k):
28          if k == len(primes):
29              yield 1
30          else:
31              rest = generate(k+1)
32              prime = primes[k]
33              for factor in rest:
34                  prime_to_i = 1
35                  # prime_to_i iterates prime**i values,
36                  # i being all possible exponents
37                  for _ in range(factors[prime] + 1):
38                      yield factor * prime_to_i
39                      prime_to_i *= prime
40
41      # in python3, `yield from generate(0)` would also work
42      for factor in generate(0):
43          yield factor
44
45
46  def distribution(name,test):
47
48      """
49      Creates csv file with the number tested, the bit range and
```

```
    the amount of divisors in that bit range
    """
    import csv

    # create csv file and add header
    with open(name+".csv", mode='w') as f:
        f_w = csv.writer(f, delimiter=',', quotechar='"',
                         quoting=csv.QUOTE_MINIMAL)
        f_w.writerow(["num", "bits", "divisors"])


    # range of values we are testing
    for num in test:

        # translate divisors to their bit length
        bit_length = []
        for i in list(divisors(num)):
            bit_length.append(i.bit_length())

        # count how many in each bit range
        for i in range(1, num.bit_length() + 1):

            with open(name+".csv", mode='a') as f:
                f_w = csv.writer(f, delimiter=',', quotechar='"',
                                 quoting=csv.QUOTE_MINIMAL)
                f_w.writerow([num,i,
                    bit_length.count(i)/(2**i - 2**(i-1))])


# list form: https://oeis.org/A002201/b002201.txt
SHCN =[2,
       6,
       12,
       60,
       120,
       360,
       2520,
       5040,
       55440,
       720720,
       1441440,
       4324320,
       21621600,
       367567200,
       6983776800,
       13967553600,
```

```
        321253732800,
        2248776129600,
        65214507758400,
        1956435232752000,
        60649492215312000,
        1212989844306240000,
        44880624239330880000,
        1840105593812566080000,
        79124540533940341440000,
        3718853405095196047680000,
        185942670254759802384000000,
        9854961523502269526352000000,
        581442729886633902054768000000,
        116288545977326780410953600000,
        12791740057505945845204896000000,
        780296143507862696557498656000000,
        23408884305235880896724959680000000,
        1568395248450804020080572298560000000,
        111356062640007085425720633197760000000,
        8128992572720517236077606223436480000000,
        642190413244920861650130891651481920000000,
        83484753721839712014517015914692649600000000,
        6929234558912696097204912320919489916800000000]

test = range(1, 18694273 + 1)

distribution("divisors",test)

distribution("HCN",SHCN)
```

B.4 EquSLPTest.py

```
#!/usr/bin/env python3

"""
EquSLPTest is used to calculate the performance
of the EquSLP algorithm.
"""

def EquSLP(X,Y,R):

```

```
    """
    EquSLP is used to decide if two Straight Line Programs are
    equal by simply calculating each arithmetic operation of each
    SLP modulo R. If both SLPs have the same modulo R, then they
    might be equal. This process is run many times with different
    R values, if it is always equal, we can conclude that both SLPs
    are equal with high probability and return True; otherwise,
    return False.

    Input: two SLPs X and Y. All SLPs start with 1.
    If X is x0 = 1, x1 = x0 + x0 and x2 = x1 * x1
    then X = [1, (0,0,"+"), (1,1,"*")].
    """

    # first gate is always 1
    modX = [1] # list of all the gates in X mod R
    modY = [1] # list of all the gates in Y mod R


    # calculate the mod of each gate in X
    for i in range(1,len(X)):

        if X[i][2] == "*":

            modX.append( ( modX[X[i][0]] * modX[X[i][1]] ) % R )

        elif X[i][2] == "+":

             modX.append( ( modX[X[i][0]] + modX[X[i][1]] ) % R )


    # calculate the mod of each gate in Y
    for i in range(1,len(Y)):

        if Y[i][2] == "*":

            modY.append( ( modY[Y[i][0]] * modY[Y[i][1]] ) % R )

        elif Y[i][2] == "+":

             modY.append( ( modY[Y[i][0]] + modY[Y[i][1]] ) % R )


    # the last gate mod R is equal to the whole SLP mod R
    if modX[-1] == modY[-1]:
        return True
```

```
56      else:
57          return False
58
59
60
61  def BitSpeed(X,Y,k):
62
63      """
64      Tests if diferent size R impacts performance.
65      Creates csv file with the input lenght, the bit range and
66      the run time.
67      """
68
69      import csv
70      from random import randint
71      import time
72
73      runs = 1000 # number of times to run test in specific range
74
75      # create csv file and add header
76      with open("BitSpeed"+str(k)+".csv", mode='w') as f:
77          f_w = csv.writer(f, delimiter=',', quotechar='"',
78                           quoting=csv.QUOTE_MINIMAL)
79          f_w.writerow(["len", "bits", "seconds"])
80
81
82      maxi = max(len(X), len(Y)) - 2
83      if maxi > 9:
84          maxi = 9
85
86      for i in range(1, 2**maxi + 1):
87
88          timer = time.time()
89
90          for _ in range(runs):
91
92              R = randint(2**(i - 1), 2**(i) - 1)
93              EquSLP(Y,X,R)
94
95          seconds = time.time() - timer
96
97          with open("BitSpeed"+str(k)+".csv", mode='a') as f:
98              f_w = csv.writer(f, delimiter=',', quotechar='"',
99                               quoting=csv.QUOTE_MINIMAL)
100             f_w.writerow([len(X)+len(Y), i, seconds / runs])
101
```

```
102
103 # X = 2^(2^8)
104 X = [1, (0,0,"+"), (1,1,"*"), (2,2,"*"), (3,3,"*"), (4,4,"*"), (5,5,"*"),
105      (6,6,"*"), (7,7,"*"), (8,8,"*")]
106
107 # Y = 2^(2^8)
108 Y = [1, (0,0,"+"), (1,1,"*"), (2,2,"*"), (3,3,"*"), (4,4,"*"), (5,5,"*"),
109      (6,6,"*"), (7,7,"*"), (8,8,"*")]
110
111 BitSpeed(X,Y,1)
112
113 # X = 2^(2^100)
114 X = [1, (0,0,"+"), (1,1,"*"), (2,2,"*"), (3,3,"*"), (4,4,"*"), (5,5,"*"),
115      (6,6,"*"), (7,7,"*"), (8,8,"*"), (9,9,"*"), (10,10,"*"), (11,11,"*"),
116      (12,12,"*"), (13,13,"*"), (14,14,"*"), (15,15,"*"), (16,16,"*"),
117      (17,17,"*"), (18,18,"*"), (19,19,"*"), (20,20,"*"), (21,21,"*"),
118      (22,22,"*"), (23,23,"*"), (24,24,"*"), (25,25,"*"), (26,26,"*"),
119      (27,27,"*"), (28,28,"*"), (29,29,"*"), (30,30,"*"), (31,31,"*"),
120      (32,32,"*"), (33,33,"*"), (34,34,"*"), (35,35,"*"), (36,36,"*"),
121      (37,37,"*"), (38,38,"*"), (39,39,"*"), (40,40,"*"), (41,41,"*"),
122      (42,42,"*"), (43,43,"*"), (44,44,"*"), (45,45,"*"), (46,46,"*"),
123      (47,47,"*"), (48,48,"*"), (49,49,"*"), (50,50,"*"), (51,51,"*"),
124      (52,52,"*"), (53,53,"*"), (54,54,"*"), (55,55,"*"), (56,56,"*"),
125      (57,57,"*"), (58,58,"*"), (59,59,"*"), (60,60,"*"), (61,61,"*"),
126      (62,62,"*"), (63,63,"*"), (64,64,"*"), (65,65,"*"), (66,66,"*"),
127      (67,67,"*"), (68,68,"*"), (69,69,"*"), (70,70,"*"), (71,71,"*"),
128      (72,72,"*"), (73,73,"*"), (74,74,"*"), (75,75,"*"), (76,76,"*"),
129      (77,77,"*"), (78,78,"*"), (79,79,"*"), (80,80,"*"), (81,81,"*"),
130      (82,82,"*"), (83,83,"*"), (84,84,"*"), (85,85,"*"), (86,86,"*"),
131      (87,87,"*"), (88,88,"*"), (89,89,"*"), (90,90,"*"), (91,91,"*"),
132      (92,92,"*"), (93,93,"*"), (94,94,"*"), (95,95,"*"), (96,96,"*"),
133      (97,97,"*"), (98,98,"*"), (99,99,"*"), (100,100,"*")]
134
135 # Y = 2^(2^100)
136 Y = [1, (0,0,"+"), (1,1,"*"), (2,2,"*"), (3,3,"*"), (4,4,"*"), (5,5,"*"),
137      (6,6,"*"), (7,7,"*"), (8,8,"*"), (9,9,"*"), (10,10,"*"), (11,11,"*"),
138      (12,12,"*"), (13,13,"*"), (14,14,"*"), (15,15,"*"), (16,16,"*"),
139      (17,17,"*"), (18,18,"*"), (19,19,"*"), (20,20,"*"), (21,21,"*"),
140      (22,22,"*"), (23,23,"*"), (24,24,"*"), (25,25,"*"), (26,26,"*"),
141      (27,27,"*"), (28,28,"*"), (29,29,"*"), (30,30,"*"), (31,31,"*"),
142      (32,32,"*"), (33,33,"*"), (34,34,"*"), (35,35,"*"), (36,36,"*"),
143      (37,37,"*"), (38,38,"*"), (39,39,"*"), (40,40,"*"), (41,41,"*"),
144      (42,42,"*"), (43,43,"*"), (44,44,"*"), (45,45,"*"), (46,46,"*"),
145      (47,47,"*"), (48,48,"*"), (49,49,"*"), (50,50,"*"), (51,51,"*"),
146      (52,52,"*"), (53,53,"*"), (54,54,"*"), (55,55,"*"), (56,56,"*"),
147      (57,57,"*"), (58,58,"*"), (59,59,"*"), (60,60,"*"), (61,61,"*"),
```

```
      (62,62,"*"), (63,63,"*"), (64,64,"*"), (65,65,"*"), (66,66,"*"),
      (67,67,"*"), (68,68,"*"), (69,69,"*"), (70,70,"*"), (71,71,"*"),
      (72,72,"*"), (73,73,"*"), (74,74,"*"), (75,75,"*"), (76,76,"*"),
      (77,77,"*"), (78,78,"*"), (79,79,"*"), (80,80,"*"), (81,81,"*"),
      (82,82,"*"), (83,83,"*"), (84,84,"*"), (85,85,"*"), (86,86,"*"),
      (87,87,"*"), (88,88,"*"), (89,89,"*"), (90,90,"*"), (91,91,"*"),
      (92,92,"*"), (93,93,"*"), (94,94,"*"), (95,95,"*"), (96,96,"*"),
      (97,97,"*"), (98,98,"*"), (99,99,"*"), (100,100,"*")]

BitSpeed(X,Y,2)



def SLPSpeed(maxlen,bits,k):

    """
    Tests if diferent size of X and Y impacts performance.
    Creates csv file with the input lenght, the bit range and
    the run time.
    """

    import csv
    from random import randint
    import time

    runs = 1000 # number of times to run test in specific range

    # create csv file and add header
    with open("SLPSpeed"+str(k)+".csv", mode='w') as f:
        f_w = csv.writer(f, delimiter=',', quotechar='"',
                         quoting=csv.QUOTE_MINIMAL)
        f_w.writerow(["len", "bits", "seconds"])

    X = [1, (0,0,"+")]
    Y = [1]

    for i in range(1, maxlen + 1):

        X.append((i,i,'*'))

        timer = time.time()

        for _ in range(runs):

            R = randint(2**(bits - 1), 2**(bits) - 1)
            EquSLP(Y,X,R)
```

```
194
195         seconds = time.time() - timer
196
197         with open("SLPSpeed"+str(k)+".csv", mode='a') as f:
198             f_w = csv.writer(f, delimiter=',', quotechar='"',
199                              quoting=csv.QUOTE_MINIMAL)
200             f_w.writerow([len(X)+len(Y), bits, seconds / runs])
201
202
203 SLPSpeed(500,8,1)
```