4-9-2021

Combinatorial and Stochastic Approach to Parallelization of the Kangaroo Method of Solving the Discrete Logarithm Problem

Joe Klesczewski
jak5779@rit.edu

MS THESIS IN COMPUTER SCIENCE

COMBINATORIAL AND STOCHASTIC APPROACH TO PARALLELIZATION OF THE KANGAROO METHOD OF SOLVING THE DISCRETE LOGARITHM PROBLEM

April 9, 2021

Thesis Committee:

Stanisław Radziszowski Chairperson

Warren Carithers Reader

Arthur Nunes-Harwitt Observer

Joe Klesczewski

Rochester Institute of Technology
Golisano College of Computing and Information Sciences
Department of Computer Science
jak5779@rit.edu

Abstract

The kangaroo method for the Pollard's rho algorithm provides a powerful way to solve discrete log problems. There exist parameters for it that allow it to be optimized in such a way as to prevent what are known as "useless collisions" in exchange for the limitation that the number of parallel resources used must be both finite and known ahead of time. This thesis puts forward an analysis of the situation and examines the potential acceleration that can be gained through the use of parallel resources beyond those initially utilized by an algorithm so configured.

In brief, the goal in doing this is to reconcile the rapid rate of increase in parallel processing capabilities present in consumer level hardware with the still largely sequential nature of a large portion of the algorithms used in the software that is run on that hardware. The core concept, then, would be to allow "spare" parallel resources to be utilized in an advanced sort of guess-and-check to potentially produce occasional speedups whenever, for lack of a better way to put it, those guesses are correct.

The methods presented in this thesis are done so with an eye towards expanding and reapplying them to this broadly expressed problem, however herein the discrete log problem has been chosen to be utilized as a suitable example of how such an application can proceed. This is primarily due to the observation that Pollard's parameters for the avoidance of so-called "useless collisions" generated from the kangaroo method of solving said problem are restrictive in the number of kangaroos used at any given time. The more relevant of these restrictions to this point is the fact that they require the total number of kangaroos to be odd. Most consumer-level hardware which provides more than a single computational core provides an even number of such cores, so as a result it is likely the utilization of such hardware for this purpose will leave one or more cores idle.

While these idle compute cores could also potentially be utilized for other tasks given that we are expressly operating in the context of consumer-level hardware, such considerations are largely outside the scope of this thesis. Besides, with the rate of change consumer computational hardware and software environments have historically changed it seems to be more useful to address the topic on a more purely algorithmic level; at the very least, it is more efficient as less effort needs to be expended future-proofing this thesis against future changes to its context than might have otherwise been necessary.

Contents

1 Introduction **5**

2 Background **6**
2.1 The Discrete Log Problem 6
2.2 Brute Force Solutions and the Birthday Paradox 7
2.3 Index Calculus (And Number Field Sieves) 9
2.3.1 Overview 9
2.3.2 Formalization 9
2.4 Prior Work on Discrete Log and its Parallelization 10
2.4.1 Pollard's Rho 10
2.4.2 Kangaroos 11
2.4.3 Useless Collisions 11
2.4.4 Pollard's Kangaroo Parameters 12
2.4.5 Extension of Kangaroos Beyond Pollard's Rho 12
2.5 Difficulties of Parallelization of Sequential Algorithms 12

3 Thesis Goals and Overview **13**
3.1 Combinatorial Analysis 13
3.2 Exploiting Combinatorial Analysis for Parallelism 15
3.3 Algorithms to Parallelize Via Combinatorial Analysis 15
3.4 Pollard's Rho Algorithm 16
3.4.1 Steps of Pollard's Rho Algorithm 17
3.5 Combinatorial Analysis of Pollard's Rho 18
3.6 Parallelization of Pollard's Rho 19
3.7 The Addition of Kangaroos to Pollard's Rho 20
3.7.1 Performance of the Kangaroo Algorithm 21
3.7.2 Steps of the Kangaroo Algorithm 22
3.7.3 Useless Collisions, Mathematically 23
3.8 Collision Detection 25
3.8.1 Floyd's Cycle Finding Algorithm vs. Distinguished Points 25
3.8.2 Floyd's Cycle Finding Algorithm 25
3.8.3 Distinguished Points 26
3.9 Parallelization of Kangaroos 27
3.10 Pollard's Parameters 28
3.11 Spare Parallel Resources 29

4 Methodology **29**
4.1 Overview 29
4.2 Hardware Choices 30

5 Results **31**
5.1 Pollard's Rho . . . 31
5.2 Single-Threaded Kangaroos . . . 32
5.3 Multi-Threaded Kangaroos . . . 36

6 Hypothesis **36**

7 Implementation Details **38**
7.1 Pollard's Rho Algorithm . . . 38
7.2 Kangaroo Method . . . 38
7.3 Generation of Discrete Log Problems . . . 39
7.4 Primality Testing . . . 39
7.5 Choice of Generator . . . 40

8 Future Work **41**
8.1 SHA-2 . . . 41
8.2 A* . . . 42

9 References **43**

Appendices **46**

A Simplified Implementations of Pollard's Rho **46**
A.1 Solving Discrete Log Problems: rho_log.cpp . . . 46
A.2 Factoring: rho_product.cpp . . . 48

B Singlethreaded Multiprecision Pollard's Rho Algorithm **49**

C Singlethreaded Multiprecision Kangaroo Algorithm **50**

D Multithreaded Kangaroo Algorithm **53**

E DLP Answer Verification Tool **59**

F DLP Generation Tool **60**

G DLP Problems **63**
G.1 8 Bit Prime Modulus . . . 63
G.2 16 Bit Prime Modulus . . . 63
G.3 32 Bit Prime Modulus . . . 64
G.4 34 Bit Prime Modulus . . . 64
G.5 36 Bit Prime Modulus . . . 65
G.6 38 Bit Prime Modulus . . . 65

G.7 40 Bit Prime Modulus 66
G.8 48 Bit Prime Modulus 66
G.9 56 Bit Prime Modulus 67
G.10 64 Bit Prime Modulus 67
G.11 72 Bit Prime Modulus 68
G.12 80 Bit Prime Modulus 68

1 Introduction

The discrete log problem underlies much of modern cryptography. Since there exist groups for which the problem is difficult but the inverse problem (discrete exponentiation) is not, the discrete log problem is suitable for use as a basis for a one-way function. Data transformations which rely on such a function therefore are easy to perform but not easy to reverse without already knowing additional information about the precise process performed.

Many algorithms have been invented for the purpose of accelerating the solution of the discrete log problem [26], including baby-step, giant-step [25], index calculus (and relatedly number field sieve) [23] [9], Pollard's rho [22], and the kangaroo method for Pollard's rho [22].

This thesis will focus on the methods by which the kangaroo method can be parallelized. Considerable previous work been done on this topic [17] [29], however recent improvements in the field of parallel hardware provide opportunities to take greater advantage of parallel algorithms than was previously feasible. Particular attention will be paid to structural elements of the kangaroo algorithm, and the combinatorial complexity thereof. Stochastic analysis of this complexity will be additionally performed where feasible, and potentially estimated where not. The goal will be to produce a clearer image of how the kangaroo method can be fruitfully parallelized utilizing modern and future hardware.

To summarize, the goals of this thesis are as follows:

- Implement Pollard's rho algorithm for solving discrete log problems (see Section 3.4 and more specifically 3.4.1)
- Implement the kangaroo method of solving discrete logarithm problems (see Section 3.7)
- Parallelize the implementation of the kangaroo method (see Section 3.9)
- Utilize Pollard's useless collision avoiding parameters (see Section 3.9)
- Take advantage of further parallel resources (see Section 3.11)

And the contributions of this thesis are as follows:

- Performance analyses of the following algorithms for solving discrete logarithm problems:
  - Single-threaded Pollard's rho algorithm (see Section 5.1)
  - Single-threaded kangaroo method (see Section 5.2)
  - Multi-threaded kangaroo method (see Section 5.3)
- Discussion of the behavior of these algorithms with regards to consumer-level hardware, including the exploitation of underutilized or otherwise nonoptimal parallel architectures

2 Background

2.1 The Discrete Log Problem

As mentioned above, the discrete log problem underlies much of modern cryptography. Classically (and with some simplification) the problem can be said to be one where, given two integers a and b relatively prime to some third integer n, we wish to find the exponent x to which one must raise a to obtain a number equivalent to b modulo n. Formally as stated in, for instance, [2]:

$$a^x \equiv b \pmod{n} \tag{1}$$

More relevant to the topic of modern cryptography, however, is another version of the above definition. Under this definition [28], we say that a discrete log problem is one where a is specified to be a generator g of the finite cyclic group g, b is restated as being some element t of that group, and we wish to find the least positive integer x such that (to use the notation in [28]):

$$g^x = t \tag{2}$$

The log expression itself then takes the form:

$$\log_g t = x \tag{3}$$

This definition is certainly more general than the previous one, and it opens up a number of possibilities. Perhaps the most important of these is that of working with elliptic curves rather than just modular arithmetic [8]. Such possibilities are outside the scope of this thesis, however, and so shall be set aside for the time being.

Since we are limiting ourselves to modular arithmetic, we will choose our notation to maximize clarity and compatibility with other work in the field [2] [14] [12]. We thus arrive at the following forms:

$$g^x \equiv t \pmod{n} \tag{4}$$

$$\log_g t = x \tag{5}$$

2.2 Brute Force Solutions and the Birthday Paradox

The brute force method of solving this problem for x is obvious: an iteration is performed over all the possible values of x until a value is found such that Expression 4 holds. For groups of especially large cardinality (that is, especially large values of the above n), this is clearly impractical [26][1].

Instead of a single linear iteration over the entire group, however, we can recast the problem in terms of finding a repeated element while walking a random (although in this case fixed) path through the group. This enables us to instead solve the problem in an expected time of just the square root of the group's size.

This property of probably finding a collision in the square root of the group size may be familiar as the key observation underpinning the "birthday paradox". The eponymous example usually given to explain the paradox is that you only need about as many people as the square root of the number of days in a year to be able to expect that at least one pair exists among them with the same birthday. Rounding up for simplicity's sake, that number is 23[2] and can be obtained from the following theorem where we are looking for the sample size n needed for us to have probability p of finding a duplicate if the number of possible values is N:

$$n(N, p) = \sqrt{-2N \ln p} + 0.5 \longrightarrow n(N, 0.5) \approx \sqrt{N} \tag{6}$$

This is only true for certain degrees of approximation, of course - the key takeaway is more that n scales with $\sqrt{N}$. Concretely for the case of 365 days in the year and a probability cutoff of 0.5:

$$\begin{aligned} n(365, 0.5) &= \sqrt{-2 * 365 * \ln 0.5} + 0.5 \\ &= \sqrt{-2 * 365 * -0.6931} + 0.5 \\ &= \sqrt{1.386 * 365} + 0.5 \\ &= \sqrt{505.9} + 0.5 \\ &= 22.49 + 0.5 \\ &= 22.99 \Longrightarrow 23 \end{aligned}$$

[1] Where exactly the line is drawn will, of course, vary. In the context of modern cryptography even values of n requiring hundreds of bits to express are considered small, and those are *certainly* impractical to approach with a brute-force approach

[2] A number which, due to being about the number of kids in many grade school classrooms, has resulted in this being the first introduction of many people to the interesting ways statistics can produce unexpected results

Or put another way, it will require 23 random samples before the probability that you have had a collision between two or more of those samples exceeds 0.5.

Formalizing instead that probability p for a group of people of size n of there being at least one pair of people with the same birthday [7]:

$$p(n) = \frac{\Pi_{i=1}^{n-1}(365-i)}{365^{n-1}} \tag{7}$$

This is then easily extended as follows [7] regarding "near miss" coincidences, wherein the probability of finding a pair with birthdays within k calendar days in a group of n people is as follows:

$$p(n,k) = \frac{(365-nk-1)!}{(365-n(k+1))!365^{n-1}} \tag{8}$$

One can further generalize to express the same probability for any number m of possible (and equally likely) birthdays:

$$p(m,n,k) = \frac{(m-nk-1)!}{(m-n(k+1))!m^{n-1}} \tag{9}$$

This problem is also known as the *simple* birthday paradox, to distinguish it from the *strong* birthday paradox [6]. In brief, the strong birthday paradox deals with the likelihood of a group of people of size n all sharing their birthday with at least one other person in the group. Mathematically, given m the number of possible birthdays and a group of people of size n, the probability that N of them will share their birthday with no other is as follows [6]:

$$p(N=k) = \Sigma_{i=k}^{n}(-1)^{i-k}\frac{i!}{k!(i-k)!}\frac{m!n!(m-i)^{n-1}}{i!(m-i)!(n-i)!m^n} \tag{10}$$

More simply for the case where $m = 365$ and $N = 0$ (that is, the case involving all the days of the year and there being nobody in the group with the same birthday as another):

$$p = \Sigma_{i=0}^{n}(-1)^{i}\frac{365!n!(365-i)^{n-1}}{i!(365-i)!(n-i)!365^n} \tag{11}$$

2.3 Index Calculus (And Number Field Sieves)

2.3.1 Overview

Some time should be set aside to discuss the current state of the art in solving discrete log problems: index calculus. More accurately, one would say that from index calculus can be derived a family of algorithms known as "number field sieve" algorithms. Index calculus operates in a probabilistic fashion, with its primary operation being the observation of discrete logs of smaller primes that are themselves more easily computed. These computations are then leveraged to express the originally desired discrete log. This approach may be easily observed as being very compatible with dynamic programming methods. More relevantly in recent years, however, most of it is also highly compatible with parallel computation. Indeed, several stages of the algorithm are what is known as "embarrassingly parallel", being able to be executed in complete parallel with themselves. The "third stage" of the algorithm doesn't rely at all on the first two and therefore may even be executed in parallel with them. The second stage is not easily parallelizable, however, which is a significant downside.

For large primes especially, this enables truly enormous speedups so long as one can supply additional hardware to the system. Supercomputers, essentially. Such an approach is therefore far less useful on consumer hardware which is often comparatively lacking in terms of the number of available computational cores[3]. Additionally, significant adaptation of the principles of number field sieve algorithms is required for such to work efficiently on groups defined by way of elliptic curve rather than modular arithmetic. This is because[4] the former groups lack as straightforward a concept of prime elements as the latter ones possess.

2.3.2 Formalization

The equation we are to solve is as given in Equation 4 and is repeated here:

$$g^x \equiv t \pmod{n}$$

We are given a generator g, the target number t, and the modulus n. From these we are asked to find the exponent x to which g is raised in order to obtain a number equivalent to $t \pmod{n}$. In other words, we are asked to find the discrete log base g of t mod n.

[3] though it should be noted that modern GPUs go a fairly long way towards providing large quantities of additional compute cores on consumer hardware

[4] among other impediments

The algorithm is, as alluded to above, performed in three stages. The first two find the discrete logarithms of a "factor base" of r smaller primes, often chosen to be first r primes (starting with the number 2) in addition to negative one, given both g and n. The third stage finds the discrete log of t in terms of these discrete logs of the factor base. In greater detail:

The first stage consists of the construction of a system of linear equations in r variables. To do this, we search for r linearly independent relations between the factor base and powers of g. Each equation is the discrete logarithm of one of the primes in the factor base. Since each of these operations is fully independent, the first stage my be executed fully in parallel with itself using r computational cores.

The second stage solves the system of linear equations that results from the first stage. If r is large (millions, perhaps), then this step is prohibitively computationally expensive on all but the most advanced hardware as it is not easy to parallelize[5].

The third stage searches for a power y of g that, when multiplied by t, is possible to factor in terms of the factor base. This is once more easily parallelized and produces the following expression:

$$g^y t = (-1)^{f_0} 2^{f_1} \dots p_r^{f_r} \tag{12}$$

The final result then merely requires simple algebraic rearranging of the results of the second and third stages. This is occasionally referred to as a "fourth stage" by some, but it is simple enough that not all bother to do so. It produces a result matching the following expression:

$$x = f_0 \log_g(-1) + f_1 \log_g 2 + \dots + f_r \log_g p_r - s \tag{13}$$

2.4 Prior Work on Discrete Log and its Parallelization

2.4.1 Pollard's Rho

One of the leading categories of methods of solving the discrete log problem derives from a 1978 paper by John M. Pollard [22] which pioneered what became known as "Pollard's rho" algorithm for solving discrete log problems. Later developments by the same author and others produced a variation on the algorithm which came to be known as the "kangaroo" algorithm, which itself has

[5]Specifically, hardware with extremely high per-core processing speed is what is required here

been refined over the years [24] [29].[6]

2.4.2 Kangaroos

A particularly useful overview of the historical developments of this algorithm can be found in a 2003 paper by Edlyn Teske [29]. The paper covers in great detail the broad category of algorithms that have developed from Pollard's rho over the decades, dividing them into three categories: the "rho method", the "kangaroo method", and the "parallelized rho" method. The paper also discusses the parallel form of the kangaroo method, but of the two only the parallel form of the rho method is called out separately since the former works in a fundamentally similar manner to the sequential kangaroo method while the latter works in a fundamentally different method to the sequential rho method. Figure 1 of Teske's paper is recommended as a useful illustration of each of the three categories.

A more detailed treatment of the specifics of the parallel form of the kangaroo method of solving discrete log problems can be found in Section 3.9, but a concept from that section must be mentioned here for the sake of clarity moving forward. The aforementioned section discusses the idea of "herds" of kangaroos, divided between a "wild" herd and a "tame" herd. Briefly put, the goal of the parallel kangaroo method is to provoke a collision between kangaroos, one being of the "wild" herd and the other being of the "tame" herd[7].

2.4.3 Useless Collisions

A topic that Teske spends a lot of time discussing is the wasted computations caused by what she calls "useless collisions". These are, simply put, when a collision is detected but winds up having been between two kangaroos of the same herd (i.e. a wild kangaroo colliding with a wild kangaroo or a tame kangaroo colliding with a tame kangaroo). Such collisions cannot be use to produce mathematically useful results, and as such are termed "useless". Processing these useless collisions can waste a considerable amount of time that would be preferably spend finding additional (and hopefully non-useless) collisions.

Using the simpler method of parallelizing the kangaroo method, Teske estimates that the processing time of the algorithm when utilizing small numbers of processors (four, in the example given therein) may be over 15% longer due solely to the time wasted while processing the useless col-

[6] Indeed, the basic Pollard's rho algorithm does not appear to be used for very much anymore, except perhaps as a low-complexity approach for small values.

[7] Technically speaking, this is also the goal of the sequential kangaroo method. But since the sequential version of the algorithm cannot make effective use of more than a single kangaroo of each "herd", such is not a meaningful manner of categorization or collation.

lisions. Some of this can be ameliorated by simulating multiple kangaroos on each processor, but that would have the obvious downside of reducing the tick rate of the algorithm. And while it does make useless collisions less of a problem, they still are a problem. In other words, this strategy provides only some benefit in exchange for nontrivial downside.

For a mathematical treatment of these "useless collisions", see Section 3.7.3.

2.4.4 Pollard's Kangaroo Parameters

Teske's 2003 paper [29] calls back to a very important 2000 paper by Pollard [24] that discusses a way to parallelize the kangaroo method while avoiding the problem of useless collisions. The trade off of this method is that the number of available processors needs to be both fixed and known ahead of time. Such may not be viable for consumer hardware due to frequently and unexpectedly fluctuating task loads on individual processor cores. This thesis draws upon Pollard's development of this parallelization method and seeks to make it more viable on consumer level hardware where the precise availability of the system's parallel computation resources may be variable or uncertain at the time when the algorithm begins to run.

2.4.5 Extension of Kangaroos Beyond Pollard's Rho

In a recent paper [17] the kangaroo algorithm was further extended to enable its use in executing side-channel attacks on systems whose security is based on the discrete log problem. The key concept promulgated in that paper (or at least, the key concept as far as this thesis is concerned) is that the properties of the algorithm that make it good at searching a large modular group when solving the discrete log problem are analogous to the very same properties that make such a problem difficult to solve in the first place. In the aforementioned paper, this is leveraged alongside various probability estimations to compute the rank of a key and thereby reduce attack times by a square or even cube root depending on circumstance. This thesis aims to exploit much the same core observation though in the opposite manner. In essence, the goal is to leverage similar probability estimates to allow the combinatorial exploitation of the kangaroo algorithm instead of the exploitation of a combinatoric system using kangaroos.

2.5 Difficulties of Parallelization of Sequential Algorithms

Few would dispute that consumer CPUs have advanced leaps and bounds in the number of instructions they can execute per unit of time. It should be noted, however, that of late much of this advancement has been in the form of additional parallel cores rather than increases in the proces-

sor's overall clock speed [3]. Meanwhile, software has generally not kept up with the increasing hardware lean towards parallelism, with especially major software suites placing heavy loads on one or maybe a few cores [15].

As a result, consumer CPUs often sit with some or even most of their cores near idle even as the others are running at maximum utilization. Such cases indicate software that would greatly benefit from being less sequential and more parallel [32].

That's not a trivial thing to change, however. A lot of algorithms by their very nature must be performed sequentially, if for no other reason than each step depending strongly on the result of the step immediately preceding it [11]. While these algorithms would certainly benefit greatly from being parallelized, they remain fundamentally incompatible with conventional approaches to parallelization [10] despite the field being extremely competitive and well-studied [30].

This thesis explores an unconventional approach to parallelization that seeks to add a small amount of parallelism to sequential algorithms while requiring only minor changes to the algorithm, if any. This parallelism is intended to not require the number or disposition of threads to be known ahead of time, as the goal is to enable sequential algorithms to take advantage of idle parallel resources on something of an ad-hoc basis.

3 Thesis Goals and Overview

3.1 Combinatorial Analysis

The examination of algorithms in terms of their complexity according to various metrics is a considerably well-studied field [20]. The particular metric of interest here is that of "combinatorial complexity" [21]. For our purposes, this might be considered a measure of the problem space of an algorithm and especially the dimensionality thereof. Analogies might be drawn to similar measures such as information entropy[8] or search space[9].

An example may be in order here. Consider an algorithm which when given two n bit numbers will, based on those two numbers, produce a third n bit number. There are 2^{2n} possible inputs to this algorithm, yet only 2^n possible outputs. Put another way, the input "step" of the algorithm has 2^{2n} possible "states" it can be in, while the output "step" of the algorithm has only 2^n possible "states" it can be in.

[8] generally the amount of uncertainty in some variable's possible outcomes

[9] generally the set or domain through which an algorithm searches for for a result

This concept of each "step" being characterized by the "states" it can be in is admittedly somewhat an abuse of terminology. But it makes the work to be done in this thesis far more concise to describe[10], and so its usage is hoped to be helpful in net.

Tying this back to the idea of "combinatorial complexity", we can fairly straightforwardly consider an algorithm's execution as a navigation between a finite number of finite sets of states[11]. The algorithm described above, for instance, might be expressed as a sequence of sets $S_0, S_1, S_2, ..., S_f$, wherein f is the number of steps the algorithm takes to produce a result. The cardinality of each of these sets is the number of possible states the algorithm could be in during the corresponding step.

Importantly, one can clearly observe the following:

- S_0 is of cardinality 2^{2n}, as the two input numbers have 2^n possible values each.
- S_f is of cardinality 2^n, as the single output number has 2^n possible values.
- If this is a deterministic algorithm[12], none of the sets in the sequence may have a cardinality greater than 2^{2n}, the cardinality of S_0. More generally, no set may have a greater cardinality than a set preceding it.
- Similarly, none of the sets in the sequence may have a cardinality less than 2^n, the cardinality of S_f. More generally, no set may have a lesser cardinality than a set following it.

Further developing this idea, we can express each step in the algorithm as a function mapping from one set in the sequence of sets to the following set. Defined generally, these functions can also express groups of steps. We shall call these functions $STEP_{i,j} : S_i \mapsto S_j$, where i and j are integers, $i \in [0, f)$, $j \in (0, f]$, and $i < j$.

As a consequence of the above, the fact that $STEP_{0,f}$ maps a larger set (S_0) to a smaller set (S_f) means that so too must one or more $STEP_{i,i+1}$. That is, one or more steps in the algorithm must cause the combinatorial complexity of the algorithm after the step to be lower than the combinatorial complexity of the algorithm before the step.

Whenever this property of $STEP_{i,j}$ mapping a larger set onto a smaller set is observed, the question of just how the domain is mapped onto the range becomes very interesting. In particular, one observes that every element of S_j (by definition of being an element in S_j) is always mapped to by at least one element of S_i and that the most elements of S_i that can map to the same element of S_j is $|S_i| - |S_j|$.

[10] and, happily, to write

[11] While conceptually algorithms are perfectly capable of requiring an infinite number of steps or an infinite amount of space, such cases can be considered outside the scope of this thesis by dint of such algorithms not being computable on obtainable hardware

[12] Indeed, this is an assumption we shall make herein

Building on this, if one can find cases where many similar elements of S_i map to a single element of S_j for some $STEP_{i,j}$ then one can potentially produce a heuristic that allows one to, upon reaching the i-th step of the algorithm, make an educated guess as to what the state of the algorithm will be in the j-th step of the algorithm. If $j = f$, then one is instead making an educated guess as to the final result will be before the algorithm has finished executing. Stochasically, this can produce a measurable speedup if one is performing many such operations in total[13].

3.2 Exploiting Combinatorial Analysis for Parallelism

This presents an opportunity for parallelism. Any operations that depend on the outcome of the j-th step can have work on them started in a separate thread based on the guess that the state predicted by the aforementioned heuristic will be arrived at. Meanwhile, the original thread continues as it was, and once it reaches the j-th step it either verifies that the guess was correct or reveals it to have been false.

In the first of these cases, the original thread ends and the thread spun off earlier replaces it. This results in time savings equal to the time the spun of thread has been executing between having been spun off and replacing the original thread. Which assumes the spun off thread had been executing on a core that would otherwise have been idle, of course.

In the second of these cases, the spun off thread is ended and the original thread continues to its next operation as if no thread had ever been spun off. A very tiny amount of time could potentially be lost due to additional overhead, but that is dependent on both implementation and optimization.

3.3 Algorithms to Parallelize Via Combinatorial Analysis

Not all algorithms are equally compatible with this approach, it should be noted. Most obviously, any that are already embarrassingly parallel would (for fairly obvious reasons) generally have little use for an additional, only probabilistically useful way to leverage parallel resources. Examples include the generation of images of the Mandelbrot set, numerical integration, and Monte Carlo analyses.

In searching for an algorithm that is more compatible with this approach to parallelization, one should look for one that are either purely sequential (i.e. each step of the algorithm relies on the previous) or inflexibly parallel (i.e. only a specific number of set of number of threads are effectively utilized). Examples of each of these, respectively, are Pollard's rho algorithm and

[13]The individual likelihood of any given run of the software seeing any speedup from this process may, depending on the quality of the aforementioned heuristic, be very low indeed

the kangaroo method (when parallelized using Pollard's parameters for the avoidance of "useless collisions" - see Section 2.4.3).

3.4 Pollard's Rho Algorithm

The term "Pollard's rho" generally refers to two distinct but closely related algorithms. One is used for the prime factorization of composite numbers, and the other is used for solving the discrete log problem [13]. A simple implementation of the former can be found in Appendix A.2 for the sake of illustration, but it is the latter that we are interested in here. A simple implementation of that can be found in Appendix A.1, also for the sake of illustration.

Pollard's rho algorithm is even still among the best options for solving the discrete log problem, especially for relatively small groups, due to negligible storage requirements and complexity competitive with other methods [8] [27]. These factors, perhaps needless to say, are significant upsides to the algorithm, and over the years it has been significantly refined from its original state [8] [13] [19].

Indeed, the expected number of multiplications required to solve a discrete log problem using the Pollard's rho algorithm is given by the following expression [29]:

$$\sqrt{\pi(\texttt{ord } g)/2} + O(\log(\texttt{ord } g)) \tag{14}$$

The Pollard's rho algorithm is a combination of a numerical function and a cycle-finding algorithm [8] [22]. The numerical function f is a random (or to be more specific here, psuedorandom) mapping of the elements of the aforementioned cyclic group G to themselves. Or more formally: $f : G \longmapsto G$. Starting from some initial value $x_0 \in G$, we produce a sequence x_1, x_2, x_3, etc. using the following definition:

$$x_{i+1} = f(x_i) \tag{15}$$

Because this is an infinite sequence and G is a finite group, eventually any element x_m must be repeated. That is, it must equal some later element x_k where $m < k$. In other words we can find m such that element x_m equals not merely some x_k, but an x_k such that $k = 2m$. The result of this is that x_m must equal some later element x_2m. This occurrence is called a "collision". Finding these collisions is fundamental to the algorithm. In Pollard's 1978 paper [22] he suggested the utilization of Floyd's cycle-detection algorithm for this purpose, and even today that is the approach utilized

for this algorithm.

3.4.1 Steps of Pollard's Rho Algorithm

Let G be a finite cyclic group whose order n is a prime and g, t are some elements of G such that g is a generator of G. The goal is to find some integer x such that the following holds:

$$g^x \equiv t \pmod{n} \tag{16}$$

G is then partitioned into r pairwise disjoint subsets $S_1, S_2, S_3, ..., S_r$ of at least roughly equal order. Pollard suggested [22] (and we will imitate, as it works well for this explanation) $r = 3$ as follows:

$$v(x_i) = \begin{cases} S_1 & 0 \leq x_i < n/3 \\ S_2 & n/3 \leq x_i < 2n/3 \\ S_3 & 2n/3 \leq x_i < n \end{cases} \tag{17}$$

Let $x_0 = 1$ (which, importantly, is not an element of S_2) be the initial value, and the aforementioned iterating function be as follows:

$$x_{i+1} = f(x_i) = \begin{cases} g * x_i & x_i \in S_1 \\ x_i^2 & x_i \in S_2 \\ t * x_i & x_i \in S_3 \end{cases} \tag{18}$$

As we compute each x_i, we also compute the values a_i and b_i such that $x_i = g^{a_i} * t^{b_i}$ is satisfied. Given that $x_0 = 1$, $a_0 = b_0 = 0$. Further values are then calculated as follows:

$$a_{i+1} = \begin{cases} a_i + 1 \pmod{n} & x_i \in S_1 \\ 2 * a_i \pmod{n} & x_i \in S_2 \\ a_i & x_i \in S_3 \end{cases} \tag{19}$$

$$b_{i+1} = \begin{cases} b_i \pmod{n} & x_i \in S_1 \\ 2 * b_i \pmod{n} & x_i \in S_2 \\ b_i & x_i \in S_3 \end{cases} \tag{20}$$

By use of Floyd's cycle detection algorithm we now search for the aforementioned elements x_m

and x_{2m}, resulting in the following equality:

$$x_m = g^{a_m} * t^{b_m} = g^{a_{2m}} * t^{b_{2m}} = x_{2m} \tag{21}$$

Simplifying, we get:

$$g^{a_{2m}-a_m} = t^{b_m-b_{2m}} \tag{22}$$

For the sake of clarity an important distinction needs to (or at least should) be made here. Mathematically, the next step is to take the discrete log of both sides of this equality. The idea of an algorithm to solve a discrete log problem requiring one to take two discrete logs may seem circular to some. The purpose of doing is to obtain the following equivalency:

$$(a_{2m} - a_m) \equiv (b_m - b_{2m}) * \log_g(t) \pmod{n} \tag{23}$$

In essence, what we have done is applied the equality in such a way as to "factor out" (to abuse some unrelated terminology) the actual discrete log operation. This distinction is perhaps made clearer when we examine the following step. So long as $\gcd(b_m - b_{2m}, n) = 1$, which is necessarily true if $b_m \not\equiv b_{2m} \pmod{n}$, we can find the multiplicative inverse of $b_m - b_{2m} \pmod{n}$ and multiply both sides by it to get:

$$\log_g(t) \equiv (b_m - b_{2m})^{-1}(a_{2m} - a_m) \pmod{n} \tag{24}$$

Algorithmically, neither of these two steps have actually required a discrete log to be taken. Rather, the multiplicative inverse of $b_m - b_{2m} \pmod{n}$ was found and $(a_{2m} - a_m)$ was multiplied by it. From the previous two expressions we are therefore able to conclude that the result of this process is equivalent to $\log_g(t) \pmod{n}$.

3.5 Combinatorial Analysis of Pollard's Rho

To further illustrate the concept of "combinatorial analysis" explained in Sections 3.1 and 3.2, we shall now relate the Pollard's rho algorithm that we have just explained back to it. The former of the two contains an abstracted explanation of the process that may be useful to reference.

First of all, it should be noted that Pollard's rho algorithm is generally performed with numbers

significantly larger than 16 bits. 64 bits[14] are significantly more relevant to problems of modern difficulty. Even with this increase in scale, however, the basic observations remain essentially identical. Using the same terminology as in section 3.1:

- S_0 is of cardinality 2^{128}, as each of the two input numbers have 2^{64} possible values each.
- S_f is of cardinality 2^{64}, as the single output number has 2^{64} possible values.
- Pollard's rho being a deterministic algorithm, none of the sets in the sequence may have a cardinality greater than 2^{128}, the cardinality of S_0. More generally, no set may have a greater cardinality than a set preceding it.
- Similarly, none of the sets in the sequence may have a cardinality less than 2^{128}, the cardinality of S_f. More generally, no set may have a lesser cardinality than a set following it.

The entire algorithm can be denoted as the function $STEP_{0,f}$, to continue to use the terminology established in Section 3.1. As discussed therein, this function maps a larger domain (S_0, which is of cardinality 2^{128}) to a smaller range (S_f, which is of cardinality 2^{64}).

The purpose and goal, then, of performing combinatorial analysis on the algorithm is to identify ways in which one can recognize related subsets of elements in the domain of $STEP_{0,f}$ (or, more generally, some $STEP_{i,j}$) that map to the *same* element in its range. Or at the very least largely do so, if such a tradeoff can be made to expand the size of the aforementioned subsets and/or increase the speed at which they are recognized. These subsets of S_0 (or again more generally, S_i) can subsequently be leveraged to make the aforementioned "educated guesses" as to the future state of the algorithm. See Section 3.2 for more information on that process.

3.6 Parallelization of Pollard's Rho

When parallelizing Pollard's rho algorithm, the direct approach is to generate one independent x_i sequence per parallel thread. These sequences are then iterated until two sequences are found to collide. The point at which this collision occurs is then used to find the solution to the original problem [29].

The sequential version of the Pollard's rho algorithm can be graphically represented by a figure resembling the Greek letter *rho* (ρ), hence the name. The initial element x_0 is the bottom-most point of the figure, and subsequent elements x_i are drawn upwards and around into a cycle. The solution is found at the point where the tail of the rho collides with the head of the rho.

[14]Or more! Though the exponential relationship between runtime complexity and the size (in bits) of the numbers involved means that even systems capable of easily handling 64-bit problems may be completely incapable of solving a 128-bit problem in a reasonable span of time.

This being the case, the parallelized version of the algorithm can be similarly represented by a bundle of *rhos*. However, instead of the solution being found at the point where the sequence reaches a cycle it is instead found at the point where two sequences collide - locally to this point, the two sequences form a figure resembling the Greek letter *lambda* (λ) instead. Because of this, the parallelized form of the Pollard's rho algorithm is rarely referred to as the lambda algorithm or even more rarely as Pollard's lambda algorithm.

Edlyn Teske produced an excellent diagram of this difference in structure in [29], which is reproduced in part as Figure 1 below. Note the area indicated by the dashed circle, and how the two colliding rhos form a shape resembling a lambda.

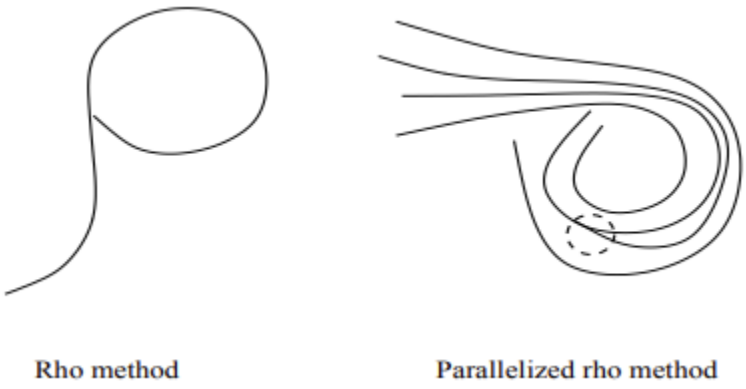

Figure 1: Teske's diagram of Pollard's rho and the parallelized variant thereof [29]

3.7 The Addition of Kangaroos to Pollard's Rho

A further development of Pollard's rho algorithm is the so-called "kangaroo algorithm" or "lambda algorithm". As the latter name can be confusing given the same name being sometimes used to refer to the parallelized variant of Pollard's rho algorithm, we will prefer to use the former of the two names in this thesis.

It is important to note here that this lambda figure describing the kangaroo algorithm is *not* the same as the lambda figure describing the solution point of the parallelized Pollard's rho algorithm. While they are not completely dissimilar in that they both describe the collision of two series and that the solution to the problem is derived from finding the collision point, the manners in which the two algorithms' lambda shapes arise are extremely different. Another part of Teske's excellent diagram in [29] is reproduced in Figure 2 below so as to illustrate this fact - compare it with the illustration of the parallelized rho method in Figure 1 above.

The key development of the kangaroo algorithm is that instead of generating a single sequence x_i

Figure 2: Teske's diagram of the kangaroo method's lambda shape [29]

that is searched until a cycle in that sequence (in other words a "collision" with itself) is found, we instead generate a pair of sequences t_i and w_i which we search for a collision between. These sequences are thought of as representing a "tame" and "wild" kangaroo respectively, hence the choice of variables. These sequences are followed until they collide, at which point the solution is found.

In actuality, these two sequences are the same sequence x_i with different start points, the sequence being x_i defined as the elements in a cyclic group G where a generator g is raised to successive exponents $\{g^0, g^1, g^2, g^3, ...\}$.

3.7.1 Performance of the Kangaroo Algorithm

The kangaroo algorithm may generally be brought up as a more performant variation of Pollard's rho algorithm, but this is not necessarily the case. In the above discussion, particularly in Equation 14, it is assumed for the sake of simplicity that the final result x might be any value on the interval [0, $\texttt{ord}\ g$]. However, it is entirely possible to limit our scope for x as instead being on some sub-interval $[a, b] \subset [0, \texttt{ord}\ g]$. Doing so yields an expected running time of:

$$2 \times \sqrt{b - a} + O(\log(\texttt{ord}\ g)) \tag{25}$$

If we allow x to be on the entire original period [0, $\texttt{ord}\ g$], then the kangaroo algorithm actually exhibits worse performance, not better. The expected running time of the kangaroo algorithm winds up being approximately 1.6 times longer than that of Pollard's rho algorithm, in fact. This effect lessens as the period $[a, b]$ shrinks. In general, the kangaroo algorithm is only faster when $b - a < \pi/8 \times (\texttt{ord}\ g)$.

In order to leverage this property, we reduce the original set of elements in our group down to a significantly smaller subset of "distinguished points" D. By only detecting collisions using these points we essentially allow the kangaroos to "overshoot" the collision point and then "backtrack" to allow us to find the final result. This produces an expected running time of:

$$2 \times \sqrt{b-a} + O(\log(\texttt{ord}\ \ D)) \tag{26}$$

3.7.2 Steps of the Kangaroo Algorithm

As mentioned above, the kangaroo algorithm derives two sequences w_i and t_i by starting each at a different initial index of the sequence $x_i \in \{g^0, g^1, g^2, g^3, ...\}$. Each time step sees a kangaroo hop forward by some distance s selected from a small set of r possible hop sizes S. Formally speaking: some hash function $h : G \longmapsto S$ is defined, generally by way of a numerical hash function to produce some index on the interval $[0, r]$ into a stored index of hop lengths. This can also be seen as partitioning G into r pairwise distinct subsets in a manner analogous to that performed during the course of Pollard's rho algorithm as described in section 3.4.1. Where the analogy breaks down, however, is that where Pollard's rho algorithm partitions G in order to facilitate a pseudorandom walk of the group, the kangaroo algorithm utilizes small jumps to gradually proceed through the sequence. Leaning once more on Teske's 2003 paper [29], Figure 3 below illustrates the difference in how these two algorithms cover the search space.

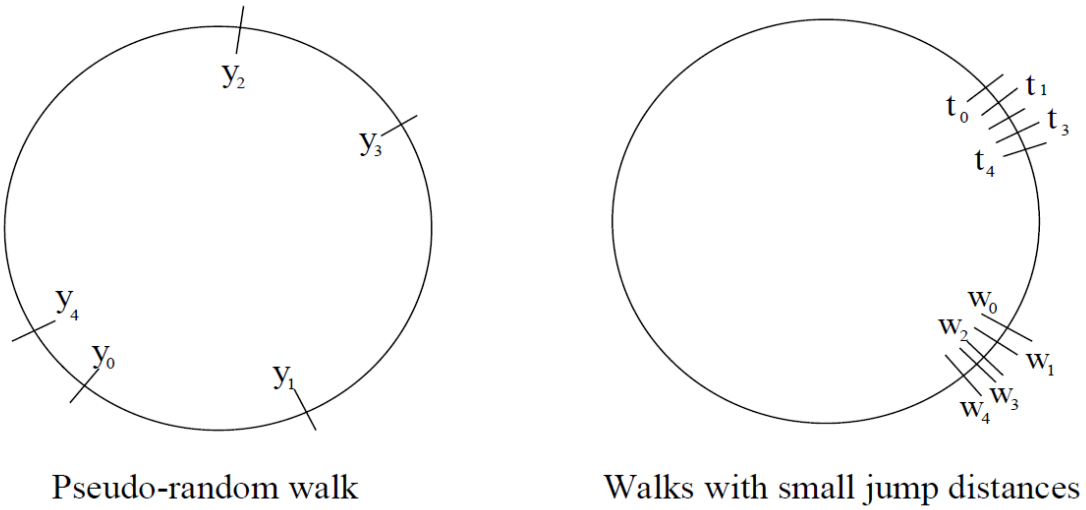

Figure 3: Teske's diagram of the kangaroo and rho methods' walks over the cyclic group G [29]

Importantly, the choice of hop distance is based solely on the kangaroo's current location, and as a result any kangaroo which lands on a particular element will proceed along a knowable path from

that point.

On the topic of the wild versus tame kangaroos, the primary difference is that the tame kangaroos start at known locations, generally evenly distributed throughout the g^i sequence. When there is only a single tame kangaroo, it would thus start at $g^{|G|/2}$. Wild kangaroos, meanwhile, "start" at some unknown element g^x - that is, they start at the target element t which we are attempting to find the discrete logarithm of[15]. The starting position exponent x is unknown, and is answer to the discrete logarithm problem that is being solved.

The tame and wild kangaroos are each run a step at a time in alternation, with the kangaroos each keeping track of their current positions (w_i and t_i) and cumulative distances traveled (d_w and d_t).

Ultimately the goal is for the wild kangaroo to land on the tame kangaroo's path - or in other words, for the path of a wild kangaroo to collide with the path of a tame kangaroo. Mathematically:

$$w_i = t_j \tag{27}$$

When this happens a solution can be derived from the knowledge of what the tame kangaroo's exponent was at the point of collision ($|G|/2 + d_t$) combined with how far each kangaroo had traveled:

$$\log_g(t) \equiv |G|/2 + d_t - d_w \pmod{n} \tag{28}$$

And therefore:

$$x \equiv |G|/2 + d_t - d_w \pmod{n} \tag{29}$$

3.7.3 Useless Collisions, Mathematically

Having thus established the mathematical and algorithmic basis of the effectiveness of the kangaroo algorithm, it is useful to once more discuss the "useless collisions" introduced in Section 2.4.3. Specifically, we established in the previous section that the final calculation required to compute the sought exponent x (see Equation 29).

With d_t and d_w representing the distances traveled by the kangaroos involved in the detected collision. This being a useful collision, one of these kangaroos is from the tame herd and the other is from the wild herd. However, we can also consider the case of a useless collision - that equation

[15] See e.g. Equation 4

would then become one of the following:

$$x \equiv |G|/2 + d_{w_i} - d_{w_j} \pmod{n} \tag{30}$$
$$x \equiv |G|/2 + d_{t_i} - d_{t_j} \pmod{n} \tag{31}$$

In the first ("wild") case, we can observe that the pairs of positions (w_i and w_j) can be re-expressed and simplified. Namely:

$$w_i \equiv x \times g^{d_{w_i}} \pmod{n} \tag{32}$$
$$w_j \equiv x \times g^{d_{w_j}} \pmod{n} \tag{33}$$

Since the nature of two kangaroos colliding allows us to observe that $w_i = w_j$, we can derive:

$$\begin{aligned} w_i &= w_j \\ x \times g^{d_{w_i}} &= x \times g^{d_{w_j}} \\ g^{d_{w_i}} &= g^{d_{w_j}} \end{aligned}$$

Which is spectacularly unhelpful. The problem is that the cancellation of x from both sides of the equations indicates that we have learned nothing about x's value.

The second ("tame") case is not any more useful. Reiterating a similar argument as above:

$$t_i \equiv g^{N/2+d_{t_i}} \pmod{n} \tag{34}$$
$$t_j \equiv g^{N/2+d_{t_j}} \pmod{n} \tag{35}$$

Which results in the derivation:

$$\begin{aligned} t_i &= t_j \\ g^{N/2+d_{t_i}} &= g^{N/2+d_{t_j}} \\ g^{d_{t_i}} &= g^{d_{t_j}} \end{aligned}$$

From which we once again find ourselves missing any way to derive a value for x and are therefore stuck. This is why such collisions are called "useless" - they cannot be used to derive a value for x like "useful" collisions can.

3.8 Collision Detection

3.8.1 Floyd's Cycle Finding Algorithm vs. Distinguished Points

The key difference between the Pollard's rho and kangaroo algorithms for solving discrete logarithms is that the former utilizes Floyd's cycle finding algorithm to detect the sequence colliding with itself (and therefore a solution having been found) while the latter utilizes a distinguished points system to detect when two kangaroos' paths have collided (and therefore found a solution).[16]

3.8.2 Floyd's Cycle Finding Algorithm

One of, if not the greatest strengths of the Pollard's rho algorithm is that it requires extremely little computation once a cycle has been has been found in the sequence:

$$x_i = \alpha^{a_i} \beta^{b_i} \tag{36}$$

Floyd's cycle finding algorithm tasks two agents with traversing this sequence: a tortoise and a hare. The former, as its name suggests, does so slowly. Each time step will generally cause the tortoise to advance from some x_i to the subsequent x_{i+1}. The hare, meanwhile, will advance much more quickly - generally twice as quickly, from x_j to $x_2 j$.

Once the tortoise's and hare's values of x match (which can be checked via a simple equality comparison $x_i \stackrel{?}{=} x_j$), the final result can easily be calculated using the following expression:

$$(b_i - b_j) * \gamma = (a_j - a_i) \pmod{n} \tag{37}$$

Where γ is the exponent that we were trying to find and the subscripts i and j indicate the tortoise and hare respectively.

[16] As a historical aside, the earliest versions of the kangaroo algorithm did not use distinguished points. However, the addition of the distinguished points method of collision detection to the kangaroo algorithm by van Oorschot and Weiner [31] was so great an improvement that one almost never[17] finds implementations of the kangaroo algorithm that do not use the distinguished points method.

[17] In the context of real-world implementations, at any rate

Importantly, this requires very little memory. The tortoise and the hare merely need to keep track of their current values of a_i and b_i, from which x_i can be recalculated each step.

3.8.3 Distinguished Points

Meanwhile, collision detection in the kangaroo method is significantly more complicated, using for this purpose what is known as a distinguished points method. This method of collision detection very significantly reduces the number of collision checks performed over the course of the algorithm's run time.

The most central component of the distinguished points method is, fittingly enough, the eponymous distinguished points. Both this and Floyd's cycle finding algorithm deal with a sequence of points x_i, and it some subset of these points that are marked as "distinguished". For the sake of efficiency, the method by which they are distinguished should be both easily checked and requiring little memory. With each x_i representing some integer value, the usual approach is to define some bitmask that can be AND-ed with the value of the current point to determine if it is a distinguished point.

Conveniently, such masks also allow for easy control of the frequency of distinguished points. For example, bitmask of $0x1$ would distinguish half of all points - the ones for which x_i is an odd number, to be specific. A bitmask of $0x11$ would distinguish a quarter of the points and $0x111$ would distinguish an eighth. The use of these bitmasks is straightforward: the current x_i is bitwise AND-ed with the mask, then if the result of that operation is not 0 then the point is not a distinguished point.

How large a portion of the points in the sequence should be distinguished is a matter of some debate. Too many points and the algorithm's memory requirements become problematic. In the worst case of *all* points being distinguished, every step would require every kangaroo to save its current state to memory. Too few points, however, and the time between one kangaroo landing on another's path and this fact being detected grows impractically long.

To resolve this quandary, it can be helpful to consider not just the absolute number of distinguished points but more abstractly the frequency with which they occur. If one eighth of the points are distinguished then one would expect each kangaroo to land on a distinguished point about once every eight steps. Same for a sixteenth of the points being distinguished leading to approximately sixteen steps between distinguished points for each kangaroo. And so on.

Obviously, there is some benefit to reducing the number of distinguished points enough that, on average, one wouldn't generally expect more than one kangaroo to land on a distinguished point

at a time. To avoid multiple kangaroos waiting for the same memory resource, if no other reason. But that is easy to do when working with small numbers of kangaroos and is completely moot as a concern when one's kangaroos are not running in parallel.

Another, perhaps more interesting angle from which to approach this question is that of memory limitations. If one's available memory space is very small, then one should only distinguish a small portion of the points in the sequence. That way, the kangaroos will only need to store their positions a few times over. The memory sizes that qualify as "very small" here may be surprising, in fact - since a kangaroo landing on a distinguished point must save both its current exponent and distance, one may require over a thousand bits every time a kangaroo lands on a distinguished point.

Constructing a mathematically simple example, consider a cyclic group G which requires 1024 (2^{10}) bits to express. Already without accounting for a kangaroo's exponent, each time a kangaroo crosses a distinguished point a kilobit is consumed. Rounding up that and any other miscellaneous overhead, it wouldn't be unwise for a final estimation of 2048 bits - two kilobits - to be consumed each time a kangaroo lands on a distinguished point. Capping the memory that is to be used at 16 gigabits, then at most kangaroos may cross distinguished points eight million times[18]. This makes the answer rather easy - we simply distinguish eight million points and can be assured that we won't run out of memory. Then in order to make sure the distinguished points are evenly distributed across the sequence, we set our bitmask to a value equaling the number of elements in our sequence divided by eight million[19].

3.9 Parallelization of Kangaroos

One of the more useful aspects of the kangaroo algorithm (like Pollard's rho algorithm) is that it exhibits a linear speed up from parallelization [8] [29] [17].

This can be achieved by expanding our single pair of one "tame" and one "wild" kangaroo into a pair of "herds" of kangaroos. We then watch for collisions between two kangaroos of different herds. Collisions between kangaroos of the same herd do occur, however, and are called "useless" as they do not contribute to the final solution. These "useless" collisions waste processing time, and as such considerable work has been done to eliminate them. The results of doing so have either left at least some "useless collisions" in (thereby still degrading performance) or required the number of parallel threads to be both fixed and known in advance.

[18]Less, admittedly, since the rest of the program would require memory, but that can be ignored for simplicity's sake - it's not all that much memory, compared to the gigabytes being used!

[19]Which might be expected to result in a distinguished point somewhere on the order of every 3×10^{298} elements or so

The latter caveat is interesting, since it's rarely a bad assumption when dealing with dedicated computing hardware. However, under some circumstances it can be too limiting. Indeed, consumer hardware is often subjected to unexpected and inconsistent loads. Even if one simply assumes that a few cores will be available at any given time under normal loads, one certainly cannot do the same for the rest of the cores. Yet it is also not unreasonable to consider it likely that one or more cores will become at least temporarily available during the program's run time, which if it is still unused will simply idle. We refer to such a core as a "spare parallel resource"[20], and it is the goal of this thesis to exploit them.

3.10 Pollard's Parameters

As mentioned in section 2.4.4, Pollard described in [24] a set of parameters for the kangaroo method that totally eliminate the aforementioned "useless collisions". In fact, Pollard was more general than this: his observation was, specifically, that if the two herds of kangaroos are chosen to consist of U tame and V wild kangaroos such that the greatest common divisor of U and V is 1 (that is, they are relatively prime), then one will never have any useless collisions. This is due to the two herds operating in distinct residue classes from their herd-mates. So long as that is the case, kangaroos of the same herd will not collide with each other.

It is further observed that it is a good idea for $U \approx V \approx M/2$ to hold, where M is the number of computational threads, due to a wildly larger number of either type of kangaroo providing decreasing benefit over a balanced number of kangaroos. The following chart provides four example Pollard-compatible configurations, ranging from the smallest possible such configuration (three kangaroos) to a configuration to one large enough as to suggest the use of GPU computing more than CPU computing:

Comment	Total Kangaroos	Tame Kangaroos	Wild Kangaroos
Smallest Valid	3	1	2
Small Example	5	2	3
Good for 8 Cores	7	3	4
Large Example	1024	511	513

Table 1: Some examples of kangaroo herd configurations using Pollard's parameters

[20] See Section 3.11

Some additional notes on the configurations proposed in this table:

- 5 kangaroos provides room on an 8-core processor for multitasking plus parallelism
- 7 does so as well minus the room for multitasking
- 1024 would be more for a graphics card system than CPU[21]

3.11 Spare Parallel Resources

An interesting observation can be made that as a result of the recent proliferation of parallel CPUs in the consumer market[23], consumer CPUs have been increasingly able to support highly parallelized loads. This hasn't been matched by a corresponding increase in parallelization of consumer applications, and as a result it's not uncommon for one or even several of a consumer's CPU cores to remain idle much of the time. The number and status of idle cores at any given time cannot be predicted or even assumed to be constant, of course, and as such the aforementioned method of parallelization that is capable of preventing all of the "useless collisions" is unable to take advantage of these inconsistently available resources.

In a sense, while the algorithm is by all accounts a parallel one, it cannot be considered ideally so since it cannot take advantage of all of the parallel resources present. Of course, by such a metric few if any algorithms might be called ideally parallel.

In this context, this thesis may be considered an exploration of a way in which algorithms can be made to use these idle parallel resources without requiring major structural changes. This will be done in the context of the Pollard's rho algorithm and in particular the kangaroo algorithm variant of same.

4 Methodology

4.1 Overview

Any run-time performance improvement due to recent hardware improvements is likely to be due to the presence of additional parallel resources, whether general-purpose (CPU) or more specialized

[21] Granted, there do exist a number of consumer-grade CPUs that provide enough cores to handle such a load. Such components are, however, still very niche at time of writing[22].

[22] Such a component would be highly promising, in fact, and it would be interesting to see the ways in which this thesis's methods could be applied to it.

[23] Indeed, single-core processors are all but extinct when looking at new machines.

(GPU) [1] [4] [5]. In either case, though particularly the latter, the first step to exploring whether this is the case should be to identify areas of the algorithm which might benefit most from these additional resources.

This process will require analysis of the combinatorial complexity of the algorithm at various steps, particularly in regards to places where a step might have relatively fewer permutations than the preceding step. Once such steps are identified, stochastic analysis may be performed in order to determine if the probability density function across this possibility space is uniform or non-uniform. In the latter case, spare parallel resources can possibly be tasked with attempting to "pre"-compute a probable value for the following step before the current one has been completed.

For the sake of expediency and complexity, the goal for this thesis is to merely demonstrate a proof-of-concept. As such, the scope of the project can be narrowed to solely concerned with the initial and final steps, i.e. the $STEP_{0,f}$ function described in Section 3.5. However, it should still be noted that any algorithm of more than nominal complexity can itself be broken down into smaller sub-algorithms and as such it is clearly the case that the application of this process is not less applicable to such cases.

4.2 Hardware Choices

As this thesis is significantly concerned about the behavior of algorithms on parallel hardware, the topic of CPU versus GPU operation cannot be ignored.

GPU operation, in particular, offers many opportunities and challenges in the development of this manner of algorithm. Such platforms enable a great many more parallel computations to occur, which in this case would correspond to a significantly larger number of kangaroos. At the same time, though, one loses a lot of flexibility with regards to those computations. CPU-GPU communication is slow, and the prospect of sending large amounts of information back and forth along the PCIe (peripheral component interconnect express) bus in order to coordinate the steps it would require[24] was frankly found to be beyond the scope of this thesis.

CPU operation's biggest strength in this case is the greater flexibility it offers in how processes are able to interact with one another. On a fundamental level the smaller number of more powerful compute cores was a better match for the needs of this thesis than the larger number of less powerful compute cores offered by GPU operation.

As such, all data in this thesis was collected utilizing consumer-grade CPUs with eight or fewer

[24] the most important being prediction, checking, skipping, and replaying, for those curious

cores.

In summary, the choice was made to perform the experiments for the collection of the data described in Section 5 locally on systems with processors not much different from the sort one might find in a contemporary consumer desktop system. Table 2 elaborates on the other options considered and provides some of the pros and cons weighed in making this determination.

Proposed Environment	Pros	Cons
Local CPU	* More conventional environment * More memory per core	* Fewest available cores
Local GPU	* Significantly more available cores	* Less memory per core * Less flexible cores * Less conventional environment
RIT Research Computing	* *Many* available cores * More powerful cores * Much more available memory * Campus-local	* Increased iteration time over local * No immediate local insight into operation
"Open Grid" parallel computing	* Largely the same as for "RIT Research Computing"	* Largely the same, plus greater variability in operation

Table 2: Hardware environment comparison

5 Results

5.1 Pollard's Rho

The data in the "Run Time Elapsed (ms)" column of Table 3 was collected by measuring the run time (specifically, the user time) required for an implementation of Pollard's Rho (implemented as described in Section 3.4.1 above) to find the discrete log given a discrete log problem wherein the prime modulus had a particular number of bits. For each size of prime modulus, twenty problems were generated under the constraint that the group's generator was two and the prime modulus was the largest safe prime of the appropriate number of bits. The aforementioned implementation of Pollard's rho algorithm was used to solve each of these problems, and the run time ("wall clock time" herein) and iterations it required to do so were measured. The resulting twenty pairs of data points were then averaged to produce a single run time measure and a single count of the iterations required for each prime length.

More specifically, the data in the "Iterations Required" column represents the number of iterations of Pollard's rho algorithm that were required in order to find a solution. This was measured by way of an accumulator which kept track of the aforementioned value during run time and was subsequently printed to the user.

The column labeled "Scaling Factor" is unlike the others in that it is a computed column rather than a measured value. Specifically, the value in that column for a row is equal to the r-th root of the ratio of that row's value for the "Iterations Required" column over the previous row's value for that column, where r is the difference between that row's value for the "Bits for Prime Modulus" column and the previous row's value for that column. The resulting number represents the "multiplier effect" on the work required when one increases the size of the prime modulus by a single bit. Interestingly, the average of these numbers is approximately $\sqrt{2}$.

For the specific discrete log problems used, see Appendix G.

Bits for Prime Modulus	Run Time Elapsed (ms)	Iterations Required	Modulus Used	Scaling Factor
16	2	318	65267	N/A
32	43	86326	4294963787	1.42
34	80	168679	17179867547	1.40
36	175	463776	68719474427	1.66
38	272	795490	274877906243	1.31
40	549	1592219	1099511627339	1.41
48	6784	20854623	281474976704939	1.38
56	141661	434255821	72057594037925099	1.46
64	2298733	7044932184	18446744073709550147	1.42
			Average:	1.43

Table 3: Run time elapsed and iterations required vs. bits of prime modulus for Pollard's rho

Figure 4 plots this data and applies fit lines. To the "Time Elapsed" data is fit the curve $0.0045 \times 2^{0.4353x}$, which fits with an R^2 value of 0.9781. The "Iterations Required" data, meanwhile, is fit by the curve $y = 1.1893 \times 2^{0.5071x}$ for an R^2 value of 0.9995.

5.2 Single-Threaded Kangaroos

The data in Tables 4, 5, 6, and 7 below was collected similarly. A single-threaded implementation of the kangaroo algorithm (i.e. one wild and one tame kangaroo share a thread - see Section 3.7.2 above for details) was used to solve each of twenty discrete log problems that were randomly generated for a given prime modulus length. These problems were (as before) generated under the

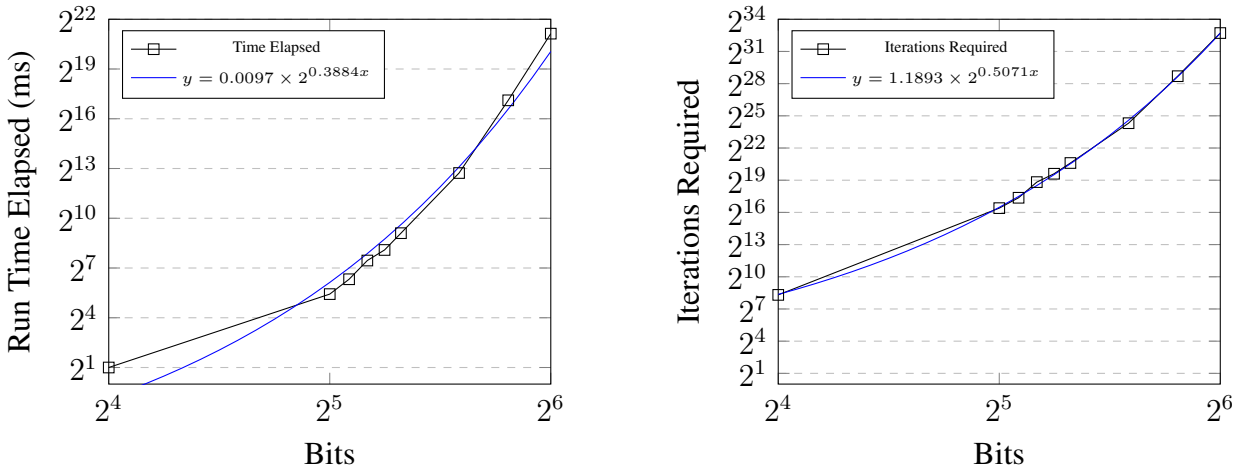

Figure 4: Graphs of Table 3

constraints of the generator having a value of two and the prime modulus being largest safe prime of the appropriate number of bits.

The maximum time allotted for any one execution of the program was generally capped at ten minutes due to schedule constraints. The moduli used were as follows:

Length in Bits	Prime Modulus
32	4294963787
34	17179867547
36	68719474427
38	274877906243
40	1099511627339
48	281474976704939
56	72057594037925099
64	18446744073709543127
72	4722366482869645207163
80	1208925819614629174700339

Bits for Prime Modulus	Time Elapsed (ms) When Mask Has Value...					Scaling Factor
	255	511	1023	2047	4095	
32	16	15	16	15	18	N/A
34	41	34	31	32	31	1.45
36	119	88	80	81	75	1.62
38	966	422	216	159	137	2.07
40	894	342	233	203	195	0.99
48	109003	58966	9581	4363	3055	1.78

Table 4: Time elapsed by single-threaded kangaroo algorithm for mask values 255-4095

Bits for Prime Modulus	Iterations Required When Mask Has Value...					Scaling Factor
	255	511	1023	2047	4095	
32	41417	41955	42634	46197	47069	N/A
34	83889	84084	84655	84655	85331	1.39
36	241399	242811	242942	245348	245582	1.70
38	524253	524448	524791	525238	525946	1.47
40	878650	878899	879303	880147	881594	1.29
48	12430628	12431446	12431193	12431713	12432224	1.39

Table 5: Iterations required by single-threaded kangaroo algorithm for mask values 255-4095

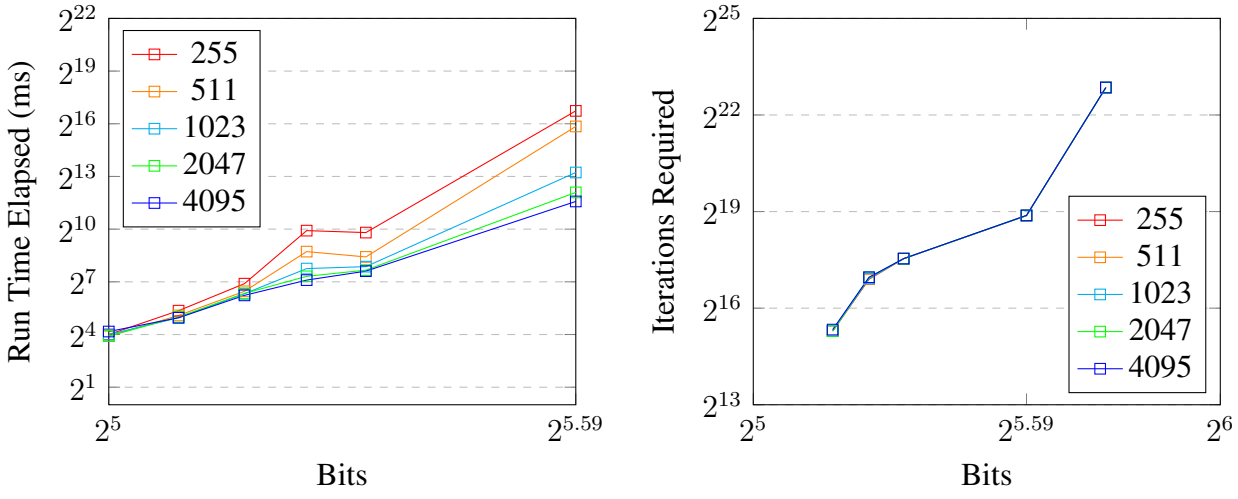

Figure 5: Graphs of Table 4 and Table 5, respectively

Bits for Prime Modulus	Time Elapsed (ms) When Mask Has Value...					Scaling Factor
	8191	16383	32767	65535	131071	
32	15	16	29	28	28	N/A
34	31	36	37	45	46	1.30
36	77	77	77	80	88	1.43
38	144	135	135	139	129	1.31
40	215	192	194	209	210	1.22
48	2639	2457	2420	2413	2421	1.36
56	45372	31185	27475	26613	26348	1.37
64	3664376	1248437	1023536	796269	734438	1.56

Table 6: Time elapsed by single-threaded kangaroo algorithm for mask values 8191-131071

Bits for Prime Modulus	Iterations Required When Mask Has Value...					Scaling Factor
	8191	16383	32767	65535	131071	
32	47361	49302	93961	93961	93961	N/A
34	90037	100987	105813	127117	127765	1.21
36	249214	251694	255507	266475	296728	1.55
38	528567	533931	539472	550009	552471	1.43
40	882422	886775	899484	932452	988977	1.30
48	12433094	12433095	12444177	12463405	12463405	1.38
56	135228745	135231442	135233688	135239296	135381908	1.35
64	1869791436	1869795850	2511748592	3250857281	3250952321	1.46

Table 7: Iterations required by single-threaded kangaroo algorithm for mask values 8191-131071

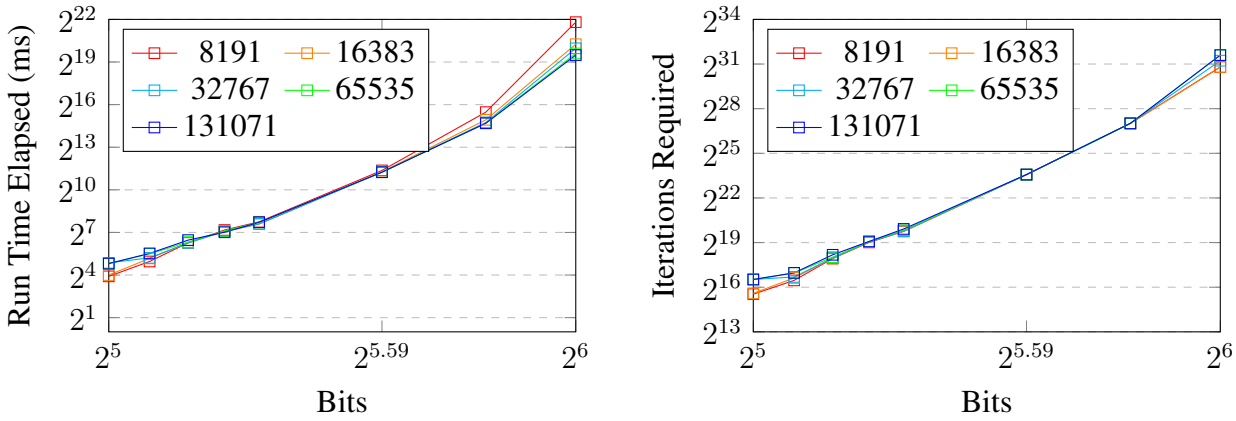

Figure 6: Graphs of Table 6 and Table 7, respectively

5.3 Multi-Threaded Kangaroos

The data in Table 8 and Table 9 was collected by this same procedure except that for both scheduling's and consistency's sakes only five runs for each combination of mask value and prime modulus size were included in the averages. Runs were capped at one hour of run time. The discrete log problems used for this were the same as those for the simple Pollard's rho algorithm implementation (see Section 5.1) and single-threaded kangaroo implementation (see Section 5.2), and can be found in Appendix G.

The code that was utilized is reproduced in Appendix D. As noted in Section 7 below, this implementation of the kangaroo algorithm is multi-threaded, currently making use of five independent threads.

Bits for Prime Modulus	Time Elapsed (ms) When Mask Has Value...				
	255	511	1023	2047	4095
32	0	1	0	0	0
34	1	1	1	1	0
36	4	4	5	3	4
38	4	5	9	6	6
40	20	14	11	11	9
48[25]	1816	713	542	412	313
56	9494	3062	2107	1701	1217
64	45702	15031	14991	14633	13539
72	1536223	269648	263047	247844	239013
80	Over 2 hr	4822982	4519909	4398777	4022640

Table 8: Parallel Kangaroo: Mask value between 255 and 4095, inclusive

6 Hypothesis

The kangaroo algorithm has a number of parameters which can be tuned to produce optimal results under specific conditions - for example, the number of kangaroos running simultaneously. The hypothesis investigated is that a combination of wild and tame kangaroo herd sizes can be identified

[25] $2^{5.59}$ in Figure 7

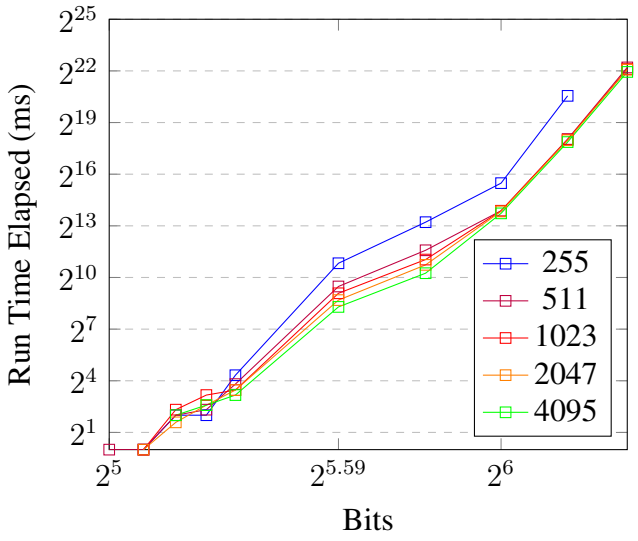

Figure 7: Graph of Table 8

Bits for Prime Modulus	Time Elapsed (s) When Mask Has Value...				
	8191	16383	32767	65535	131071
32	0	0	0	0	0
34	0	0	0	0	0
36	2	3	3	4	4
38	5	6	7	7	6
40	6	8	8	5	8
48	99	91	90	90	90
56	1055	716	635	619	609
64	13377	13217	13181	12866	11904
72	228651	218737	213382	209161	193886
80	3921356	3822622	3582411	3486404	3188283

Table 9: Parallel Kangaroo: Mask value between 8191 and 131071, inclusive

such that the run time reduction is of greater magnitude than the number of threads by itself would suggest.

In particular, the herd sizes investigated are the ones following the constraints identified by Pollard in his 2000 paper which, in exchange for requiring the number of available processors be both fixed and known in advance, totally eliminate the possibility of so-called “useless collisions” [29] [24].

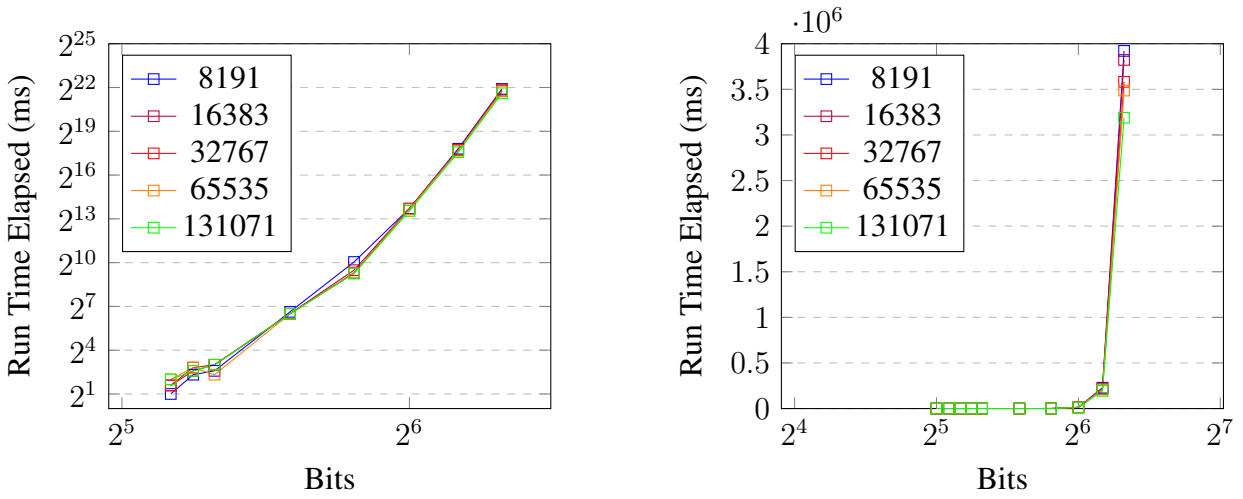

Figure 8: Graphs of Table 9, displayed logarithmically (log-log) and linearly (semilog) on the y-axis. Both are logarithmic on the x-axis.

As the number of threads utilized for the multithreaded implementation was five, that is target ratio of time elapsed for the single threaded implementation over the time elapsed for the multi-threaded implementation.

7 Implementation Details

7.1 Pollard's Rho Algorithm

The Pollard's rho algorithm and the kangaroo method were implemented as described in Section 3.4.1 and Section 3.7.2 respectively. See those sections for explanations of the algorithms or Appendix B and Appendix D (respectively) for the actual C++ code as written[26]. Additionally, simple "toy" implementations of Pollard's rho algorithm for logarithms and factoring (see Section 2.4.1 and Section 3.4) can respectively be found in Appendix A.1 and Appendix A.2.

7.2 Kangaroo Method

To produce the results in Section 5.3, this single threaded implementation of the kangaroo algorithm was re-implemented to make use of multiple processing threads, first as described in Section

[26]All code in this thesis is written in C++. The specific variety of C++ used was C++20 as compiled by GCC 8.2.0[27].

[27]To be even more specific, the options used were `-std=c++2a -O3`, plus the `-L` necessary to link the C++ Boost library and `-D`s to instantiate any preprocessor defines.

3.9 and then with the addition of Pollard's parameters for the avoidance of useless collisions as described in Section 3.10. Five threads were utilized, one per kangaroo. Two of them were earmarked for "tame" kangaroos and three were earmarked for "wild" kangaroos. The specific numbers were chosen in order to avoid the "useless collisions" described in Section 2.4.3 by relying on the parameters Pollard identified for this exact purpose, described in Section 3.10

7.3 Generation of Discrete Log Problems

Appendix F contains the code used to generate the discrete log problems found in Appendix G (details about the utilization of which can be found in Section 5). The process used to do so is as follows:

1. Determine the length, in bits, the prime modulus should be; by parameter or user input
2. Compute the largest possible number that can be stored in that many bits
3. If the current number N meets the following conditions, go to Step 4.
   Otherwise, decrement it and repeat this step, unless that would make it 0.
   (a) N is a prime number (see Section 7.4 for details).
   (b) Half of one less than N is a prime number, meaning N is a "safe prime".
   (c) N is equivalent to three modulo eight, guaranteeing 2 to be a generator of N's group.
4. The number reached in Step 3 is to be the prime modulus for the problems generated.
5. Thanks to Condition (c) of Step 3, we can simply choose 2 as the problems' generator.
6. For each desired discrete log problem, generate a random number on the interval $[1, N-1]$[28]
7. The random numbers generated in Step 5 are the problems' target numbers.

7.4 Primality Testing

Throughout this thesis's code, when the primality of a number needs to be checked the Miller-Rabin test is used. More specifically, the `boost::multiprecision::miller_rabin_test` function from the Boost C++ library is called [18].

Miller-Rabin, it is important to note, is a statistical method of primality checking that uses a random number generator[28]. It will certainly report any prime number as a prime number, but it will also report a composite number as prime with probability 0.25^{trials}. The implementation in the Boost C++ library does a little better than this due to the addition of a few checks to eliminate some possible composites, but 0.25^{trials} remains a good upper estimate of the probability regardless.

[28] A Mersenne twister pseudorandom number generator producing 64-bit numbers and with a state size of 19937 bits is used for this purpose. While plenty of alternatives do exist [26], they were not considered to be particularly beneficial in this case.

The Boost C++ library documentation, citing Knuth, recommends 25 trials for a probability[29] of approximately 9×10^{-16}.

Since this thesis deals with relatively large numbers ($2^{64} \approx 2 \times 10^{19}$ or potentially even greater) in potentially large quantities, 9×10^{-16} was considered an unacceptably high probability of failure. As such, 40 trials were performed instead for a probability of approximately 8×10^{-25}.

7.5 Choice of Generator

As mentioned above, all discrete log problems generated for this thesis were done so under the constraint of 2 being the generator. This isn't a limitation on the discrete log problem solvers, however. All three implementations (Pollard's rho, single-threaded kangaroo algorithm, and parallel kangaroo algorithm) are able to solve discrete log problems with other generators, of course. For instance, one can observe that these algorithms are capable of solving both$3^x \equiv 6 \pmod 7$ ($x = 3$) as well as $2^x \equiv 3 \pmod 5$ ($x = 3$). This is as expected given the general form of the algorithm discussed above.

This constraint was added for several reasons. Simplicity of implementation, of course, was a benefit - though as mentioned, the problem solvers were not so constrained and as such they were not made any simpler by it. But the biggest reason was to ensure that data points between implementations were as comparable as possible. Initially, it was assumed that the difference in the various implementations' performances would be great enough that a significant number of data points would be collected that would take too long to collect with at least one of the other implementations. By making all of the generators the same, that is removed as a possible source of variance in the data.

To verify that 2 is a generator (also known as a "primitive root") for each of the prime moduli, we observe that Lagrange's theorem tells us any subgroup H of a group G is of order $|H|$ such that $|G|/|H| = [G : H] \in \mathbb{Z}$. In our case of G being the multiplicative group of integers modulo a prime p, $|G| = p - 1$ due to 0 not being invertible. Therefore, any $|H|$ must be a divisor of $p - 1$. The aforementioned constraint of p being a "safe prime" (i.e. $p = 2 \times q + 1$ for some other prime q) allows us to conclude $|H| \in \{1, 2, q, p - 1\}$. We can eliminate each of these possibilities *except* for $p - 1$ as follows:

- The case of $|H| = 1$ indicates, of course, identity. 2 is not an identity element in any of the primes' groups so this possibility can be discarded.

[29] Probability of reporting a given composite number as a prime number, to be clear. It is not possible to determine the probability of a number that is reported as prime is actually prime, since the necessary prior probability is not known.

- The case of $|H| = 2$ is eliminated by observing either that such would indicate a value of -1 (which 2 is obviously not) or that Euler's criterion prohibits it by the same argument as the following point.
- As mentioned in Step 3 of the algorithm described in Section 7.3 to generate discrete log problems, the only primes that were used were safe primes which are equivalent to $3 \pmod 8$. This combined with the second supplement to quadratic reciprocity allows us to conclude $x^2 \equiv 2 \pmod p$ is not solvable, which in turn means that $2 \pmod p$ is a quadratic non-residue. From this, Euler's criterion gives us $2^{(p-1)/2} \equiv -1 \pmod p$ and consequently that 2 is not of order $\frac{p-1}{2}$.

Having eliminated all other divisors of $|G| = p - 1$, $|H|$ must therefore be $p - 1$, which is equal to $|G|$. This means that 2 is a primitive root of all our primes p and can therefore be used to generate them.

For further expansion on this line of reasoning, several existing works can be recommended such as Stinson's *Cryptography: Theory and Practice* [26] and Ireland's *A Classical Introduction to Modern Number Theory* [16].

8 Future Work

8.1 SHA-2

An analysis as described in Section 3.1 has already been completed regarding the SHA-2 algorithm. Also completed for this algorithm is the implementation of an isolated subsection of the algorithm and some work has been done to perform iterations over its possible parameters. SHA-2 is currently an extremely widespread algorithm, and as such has potentially significant upside that may warrant further investigation.

Particular attention may be given to the SHA-2 family compression function. Therein one can observe that any given iteration produces only two "new" segments - of the eight pre-iteration segment (A-H) values, six of them (B-D, F-H) are present verbatim at the start and a seventh (E) does not depend on the first four segments. Formally speaking, each of these iterations does depend on the previous. As such, the algorithm is purely sequential in nature and cannot be parallelized normally. However, exploitation of this uneven dependence may allow for some small but measurable increase in overall computation speed using the combinatoric methods described above.

Only extremely preliminary experiments have been performed, but the data so obtained has been interesting. For instance, random inputs were given to an implementation of SHA-256 such that

some arbitrarily declared value was observed early in the compression loop. Values at the end of the compression loop were then observed to see how often the same output was obtained for a different input which nevertheless caused the aforementioned arbitrary value to appear. In 300 million such computations collected over a fairly significant stretch of time, the following results were obtained:

Times a Value Appeared	Number of Values So Appearing
1	4412
2	26611
3	4801
4	10450
5	248
6	6

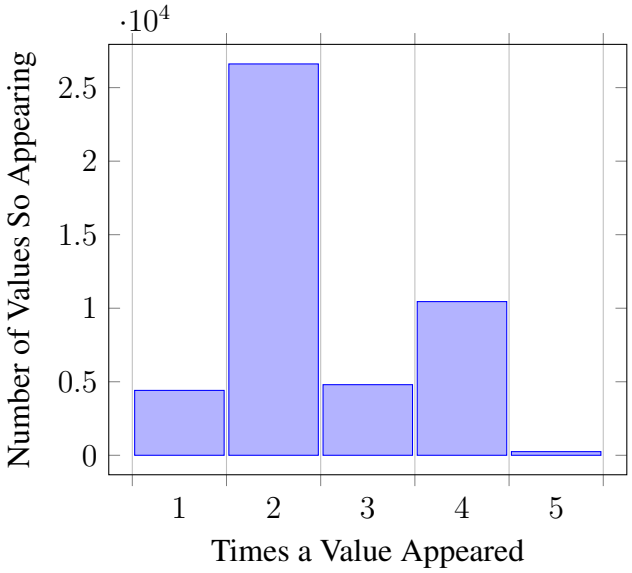

Figure 9: Preliminary SHA-256 Data

8.2 A*

It is further the case that the A* algorithm appears ripe for parallelization by this method. Current methods [10] often still involve full partition of the search space, which may be bypassed via predictive methods resultant from combinatorial analysis as described above. A* is, if anything, even more widespread than SHA-2 - similarly major upsides may be present as a result.

9 References

[1] Karam M. Abughalieh and Shadi G. Alawneh. A survey of parallel implementations for model predictive control. *IEEE Access*, 7:34348–34360, 2019.

[2] Eric Bach. Discrete logarithms and factoring. *ACM Technical Report*, 1984.

[3] Carlos Carvalho. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*, 2002.

[4] An Chen, Supriyo Datta, X. Sharon Hu, Michael T. Niemier, Tajana Šimunić Rosing, and J. Joshua Yang. A survey on architecture advances enabled by emerging beyond-cmos technologies. *IEEE Design & Test*, 36(3):46–68, 2019.

[5] Nevin Cini and Gulay Yalcin. A methodology for comparing the reliability of gpu-based and cpu-based hpcs. *ACM Computing Surveys (CSUR)*, 53(1):1–33, 2020.

[6] Mario Cortina Borja. The strong birthday problem. *Significance*, 10(6):18–20, 2013.

[7] Anirban DasGupta. The matching, birthday and the strong birthday problem: a contemporary review. *Journal of Statistical Planning and Inference*, 130(1-2):377–389, 2005.

[8] Jenny Falk. *On Pollard's rho method for solving the elliptic curve discrete logarithm problem*. Bachelor's thesis, Linnaeus University, 2019.

[9] Joshua Fried, Pierrick Gaudry, Nadia Heninger, and Emmanuel Thomé. A kilobit hidden snfs discrete logarithm computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 202–231. Springer, 2017.

[10] Alex Fukunaga, Adi Botea, Yuu Jinnai, and Akihiro Kishimoto. A survey of parallel a. *arXiv preprint arXiv:1708.05296*, 2017.

[11] Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, and Philip S. Yu. A survey of parallel sequential pattern mining. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 13(3):1–34, 2019.

[12] L. Harn. Cryptanalysis of the blind signatures based on the discrete logarithm problem. *Electronics Letters*, 31(14):1136–1137, 1995.

[13] Marcella Hastings, Nadia Heninger, and Eric Wustrow. Short paper: The proof is in the pudding. In *International Conference on Financial Cryptography and Data Security*, pages 396–404. Springer, 2019.

[14] P. Hörster, Markus Michels, and Holger Petersen. Comment: Cryptanalysis of the blind signatures based on the discrete logarithm problem. *Electronics Letters*, 31(21):1827–1827, 1995.

[15] Neng Hou, Xiaohu Yan, and Fazhi He. A survey on partitioning models, solution algorithms and algorithm parallelization for hardware/software co-design. *Design Automation for Embedded Systems*, 23(1-2):57–77, 2019.

[16] Kenneth Ireland and Michael Rosen. *A Classical Introduction to Modern Number Theory*. Springer Science Business Media, 1990.

[17] Tanja Lange, Christine Van Vredendaal, and Marnix Wakker. Kangaroos in side-channel attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 104–121. Springer, 2014.

[18] John Maddock and Christopher Kormanyos. Primality testing. http://web.archive.org/web/20180302015829/https://www.boost.org/doc/libs/1_62_0/libs/multiprecision/doc/html/boost_multiprecision/tut/primetest.html. Accessed: April 9, 2021.

[19] Hiromasa Miura, Rikuya Matsumura, Ken Ikuta, Sho Joichi, Takuya Kusaka, and Yasuyuki Nogami. A preliminary study on methods to eliminate short fruitless cycles for pollard's rho method for ecdlp over bn curves. In *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*, pages 353–359. IEEE, 2019.

[20] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[21] Leonid I. Perlovsky. Conundrum of combinatorial complexity. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(6):666–670, 1998.

[22] John M. Pollard. Monte carlo methods for index computation $(\mathrm{mod}\ p)$. *Mathematics of Computation*, 32(143):918–924, 1978.

[23] John M. Pollard. Factoring with cubic integers (the development of the nfs). In *Lecture Notes in Mathematics*, volume 1554. Springer-Verlar, New York, 1988.

[24] John M. Pollard. Kangaroos, monopoly and discrete logarithms. *Journal of Cryptology*, 13(4):437–447, 2000.

[25] Daniel Shanks. The infrastructure of a real quadratic field and its applications. In *Proceedings of the Number Theory Conference (Univ. Colorado, Boulder, Colo., 1972)*, pages 217–224, 1972.

[26] Douglas R. Stinson and Maura Paterson. *Cryptography: Theory and Practice*. Chapman and Hall, 2017.

[27] Edlyn Teske. On random walks for pollard's rho method. *Mathematics of Computation*, 70(234):809–825, 2001.

[28] Edlyn Teske. Square-root algorithms for the discrete logarithm problem (a survey). In *In Public Key Cryptography and Computational Number Theory, Walter de Gruyter*. Citeseer, 2001.

[29] Edlyn Teske. Computing discrete logarithms with the parallelized kangaroo method. *Discrete Appl. Math.*, 130(1):61–82, August 2003.

[30] Ancy Sarah Tom, Narayanan Sundaram, Nesreen K. Ahmed, Shaden Smith, Stijn Eyerman, Midhunchandra Kodiyath, Ibrahim Hur, Fabrizio Petrini, and George Karypis. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[31] Paul C. Van Oorschot and Michael J Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[32] Wei-Jie Yu, Jin-Zhou Li, Wei-Neng Chen, and Jun Zhang. A parallel double-level multiobjective evolutionary algorithm for robust optimization. *Applied Soft Computing*, 59:258–275, 2017.

Appendices

A Simplified Implementations of Pollard's Rho

A.1 Solving Discrete Log Problems: rho_log.cpp

```
#include <cstdint>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

uint16_t gcd(const uint16_t a, const uint16_t b)
{
   if(a == 0) { return b; }
   return gcd(b % a, a);
}
uint16_t inverse(const uint16_t a, const uint16_t b, const uint16_t s0
   ↪ = 1, const uint16_t s1 = 0)
{
   return b==0 ? s0 : inverse(b, a%b, s1, s0 - s1*(a/b));
}

uint16_t N = 1019, n = N-1;
uint16_t generator, target;

void new_xab(uint16_t& x, uint16_t& a, uint16_t& b) {
   switch (x % 3) {
      case 0: x = x * x % N; a = a*2 % n; b = b*2 % n; break;
      case 1: x = x * generator % N; a = (a+1) % n; break;
      case 2: x = x * target % N; b = (b+1) % n; break;
   }
}

uint16_t pollard(uint16_t g, uint16_t t)
{
   uint16_t x = 1, a = 0, b = 0;
   uint16_t X = x, A = a, B = b;
   generator = g;
   target = t;
   for(int i = 1; i < n; ++i)
   {
```

```
        new_xab(x, a, b);
        new_xab(X, A, B);
        new_xab(X, A, B);
        if(x == X) break;
    }
    if(B - b == 0 || gcd(B-b, N) != 1) { return 0; }
    uint16_t inv = (inverse(B-b, N) % n + n) % n;
    return inv * (a-A) % N;
}

int main(int argc, char **argv)
{
    if(argc != 4)
    {
        std::cout << "Usage: ###.exe <Prime N for group> <Generator g> <
            ↪ Target t>" << std::endl;
        return 2;
    }

    uint16_t g, t, res;
    N = atoi(argv[1]);
    n = N-1;
    g = atoi(argv[2]);
    t = atoi(argv[3]);
    res = pollard(g, t);

    if(res == 0) { std::cout << "FAILURE" << std::endl; }
    else { std::cout << "Result: " << res << std::endl; }
    return (res == 0) ? 1 : 0;
}
```

A.2 Factoring: rho_product.cpp

```
#include <cstdint>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

#define F(x) (x*x+1)

uint16_t gcd(uint16_t a, uint16_t b)
{
   if(a == 0) { return b; }
   return gcd(b % a, a);
}

uint16_t pollard_factor(uint16_t n)
{
   uint16_t x=2, y=2, d=1;
   while(d==1)
   {
      x = F(x) % n;
      y = F(F(x)) % n;
      d = gcd((x > y ? x-y : y-x), n);
   }

   return d != n ? d : 0;
}

// Factor a composite number
int main(int argc, char **argv)
{
   if(argc != 2)
   {
      std::cout << "Usage: ###.exe <Composite number to factor>" << std
         ↪ ::endl;
      return 2;
   }

   uint16_t res = pollard_factor(atoi(argv[1]));

   if(res == 0) { std::cout << "FAILURE" << std::endl; }
   else { std::cout << "Result: " << res << std::endl; }
   return (res==0) ? 1 : 0;
}
```

B Singlethreaded Multiprecision Pollard's Rho Algorithm

```
#include <stdio.h>

#include <boost/multiprecision/cpp_int.hpp>

#define xstr(a) str(a)
#define str(a) #a

using boost::multiprecision::cpp_int;


const cpp_int n(xstr(NORD)), N = n + 1;
const cpp_int malpha(xstr(GENERATOR));
const cpp_int mbeta(xstr(TARGET));

void new_xab(cpp_int& x, cpp_int& a, cpp_int& b) {
  switch (static_cast<int>(x % 3)) {
  case 0: x = x * x % N; a = a*2 % n; b = b*2 % n; break;
  case 1: x = x * malpha % N; a = (a+1) % n; break;
  case 2: x = x * mbeta % N; b = (b+1) % n; break;
  }
}

int main(void) {
  cpp_int x = 1, a = 0, b = 0;
  cpp_int X = x, A = a, B = b;
  for (cpp_int i = 1; i < n; ++i) {
    new_xab(x, a, b);
    new_xab(X, A, B);
    new_xab(X, A, B);
    if (x == X) break;
  }
  std::cout << x << "; " << a << "; " << b << std::endl;
  std::cout << X << "; " << A << "; " << B << std::endl;
  return 0;
}
```

C Singlethreaded Multiprecision Kangaroo Algorithm

```
#include <algorithm>
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <vector>

#include <boost/multiprecision/cpp_int.hpp>

using boost::multiprecision::cpp_int;
namespace mp = boost::multiprecision;

// Sets for step sizes
std::vector<cpp_int> S;
std::vector<cpp_int> R;

// Stuff to check distinguished points
cpp_int dpm;
bool is_dp(cpp_int x) { return (x % dpm) == 0; }

// Storage and checking of distinguished point hits
enum KType { TAME, WILD };
struct dp_hit { KType type; cpp_int x; cpp_int d; };
dp_hit collision;
std::vector<dp_hit> dp_hits;
bool was_useful_collision(KType type, cpp_int x, cpp_int d)
{
   dp_hit newhit = {type, x, d};
   for(dp_hit hit : dp_hits)
   {
      if(hit.x == newhit.x) { collision = hit; return true; }
   }
   dp_hits.push_back(newhit);
   return false;
}

long iters;
cpp_int kangaroo(cpp_int g, cpp_int t, cpp_int N)
{
   // The mean of the step sizes is approximately sqrt(N)
   while(S.size() == 0 || sqrt(N) > cpp_int(std::accumulate(S.begin(),
      ↪ S.end(), cpp_int(0)) / S.size()))
```

```
{ // All step sizes are powers of 2, starting from 1
   S.emplace_back(mp::pow(cpp_int(2), S.size()));
   R.push_back(powm(g, S.back(), N));
}

// Distinguished points mask - compile-time parameter DPMASK_VAL
#ifdef DPMASK_VAL
dpm = DPMASK_VAL;
#else
dpm = 0x11;
#endif

// Tame kangaroo
cpp_int xt = powm(g, N/2, N), dt(0);
// Wild kangaroo
cpp_int xw = t, dw(0);

bool done = false;
iters = 0;
while(! done)
{
   // Tame kangaroo hop
   size_t i = static_cast<size_t>(xt % S.size());
   xt = (xt * R[i]) % N;
   dt = dt + S[i];
   // Tame kangaroo collision check
   if(is_dp(xt))
   {
      if(was_useful_collision(TAME, xt, dt)) { return N/2 + dt -
         ↪ collision.d; }
   }

   // Wild kangaroo hop
   i = static_cast<size_t>(xw % S.size());
   xw = (xw * R[i]) % N;
   dw = dw + S[i];
   // Wild kangaroo collision check
   if(is_dp(xw))
   {
      if(was_useful_collision(WILD, xw, dw)) { return N/2 +
         ↪ collision.d - dt; }
   }

   // That's one iteration done
   ++iters;
```

```
    }

    // Return sentinel value on failure
    return cpp_int(-1);
}

int main(int argc, char **argv)
{
    // Usage check
    if(argc != 4)
    {
        std::cout << "USAGE: ###.exe <generator> <target value> <prime
            ↪ modulus>" << std::endl;
        return 2;
    }
    // Construct generator, target value, and prime modulus from the
        ↪ inputs
    cpp_int g(argv[1]), t(argv[2]), N(argv[3]);
    // Call the kangaroo algorithm and print the result
    std::cout << "Result: " << kangaroo(g, t, N) << std::endl;
    std::cout << "Required " << iters << "iterations." << std::endl;
    return 0;
}
```

D Multithreaded Kangaroo Algorithm

```
#include <algorithm>
#include <atomic>
#include <chrono>
#include <iostream>
#include <iterator>
#include <map>
#include <math.h>
#include <mutex>
#include <random>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include <thread>
#include <vector>

#include <boost/multiprecision/cpp_int.hpp>
#include <boost/multiprecision/gmp.hpp>

using boost::multiprecision::cpp_int;
namespace mp = boost::multiprecision;

const unsigned seed = std::chrono::system_clock::now().time_since_epoch
    ↪ ().count();
std::mt19937_64 generator(seed);

const int M = 5; // The total number of kangaroos
const int U = 2; // The number of tame kangaroos
const int V = 3; // The number of wild kangaroos

// Set of "jump distances" and "jumps"
// Each time a kangaroo jumps, its position is multiplied by some j_i =
    ↪ g^{s_i}
std::vector<cpp_int> S;
std::vector<cpp_int> J;

// Hash function G -> [1,r] to divide G into r pairwise disjoin subsets
int H(cpp_int G_element) { return static_cast<int>(G_element) % S.size
    ↪ (); }

// Distinguished point checking
cpp_int dpm;
bool is_dp(cpp_int x) { return (x & dpm) == 0; }
```

```
// Globals to define the DLP being solved
cpp_int N, g, t;

// Storage and checking of distinguished point hits
enum KType {TAME, WILD};
struct dp_hit {KType type; cpp_int x; cpp_int d; };
std::atomic<bool> found_useful_flag(false);
dp_hit uc_new, uc_old; // "useful collision"
cpp_int result;
std::vector<dp_hit> w_dp_hits, t_dp_hits;
bool was_useful_collision(dp_hit hit)
{
   std::vector<dp_hit> *other_herd;
   if(hit.type == TAME) { other_herd = &w_dp_hits; t_dp_hits.push_back(
      ↪ hit); }
   else { other_herd = &t_dp_hits; w_dp_hits.push_back(hit); }

   for(dp_hit &oldhit : *other_herd)
   {
      if(hit.x == oldhit.x) { uc_new = hit; uc_old = oldhit; return
         ↪ true; }
   }

   return false;
}


/* THE KANGAROOS */
std::thread threads[U+V];
struct kangaroo_data { cpp_int x; cpp_int d; };
kangaroo_data t0s[U];
kangaroo_data w0s[V];
std::atomic<bool> kangaroo_go_flag(false);


// Mutex for critical section
std::mutex mtx;

void tame_kangaroo(void *data_param)
{
   std::unique_lock<std::mutex> lck (mtx,std::defer_lock);
   // Stores current position and distance traveled - already has
      ↪ starting values
   kangaroo_data *data = (kangaroo_data *)data_param;
```

```
    #ifdef DEBUG
    std::stringstream ss;
    ss << "Tame␣kangaroo␣"
            << std::this_thread::get_id()
            << "␣constructed␣with␣parameters:"
            << "␣x=" << data->x
            << "␣d=" << data->d
            << '\n';
    std::cout << ss.str() << std::flush;
    #endif

    // Wait for start
    while(! kangaroo_go_flag) { std::this_thread::yield(); }

    while(kangaroo_go_flag)
    {
        // Jump.
        size_t i = static_cast<size_t>(data->x % S.size());
        data->x = (data->x * S[i]) % N;
        data->d += J[i];
        #ifdef DEBUG
        ss.str("");
        ss << "T" << std::this_thread::get_id() << "␣@␣(" << data->x << "
            ↪ ,␣" << data->d << ")\n";
        std::cout << ss.str() << std::flush;
        #endif

        // Check.
        if(is_dp(data->x))
        {
            lck.lock();
            if(found_useful_flag) { break; }
            if(was_useful_collision({TAME, data->x, data->d}))
            {
                kangaroo_go_flag = false;
                found_useful_flag = true;
                result = (N/2 + uc_new.d - uc_old.d) % N;
            }
            lck.unlock();
        }

        std::this_thread::yield();
    }
}
```

```
void wild_kangaroo(void *data_param)
{
   std::unique_lock<std::mutex> lck (mtx,std::defer_lock);
   // Stores current position and distance traveled - already has
      ↪ starting values
   kangaroo_data *data = (kangaroo_data *)data_param;
   #ifdef DEBUG
   std::stringstream ss;
   ss << "Wild␣kangaroo␣"
           << std::this_thread::get_id()
           << "␣constructed␣with␣parameters:"
           << "␣x=" << data->x
           << "␣d=" << data->d
           << '\n';
   std::cout << ss.str() << std::flush;
   #endif

   // Wait for start
   while(! kangaroo_go_flag) { std::this_thread::yield(); }

   while(kangaroo_go_flag)
   {
      // Jump.
      size_t i = static_cast<size_t>(data->x % S.size());
      data->x = (data->x * S[i]) % N;
      data->d += J[i];
      #ifdef DEBUG
      ss.str("");
      ss << "W" << std::this_thread::get_id() << "␣@␣(" << data->x << "
         ↪ ,␣" << data->d << ")\n";
      std::cout << ss.str() << std::flush;
      #endif

      // Check.
      if(is_dp(data->x))
      {
         lck.lock();
         if(found_useful_flag) { break; }
         if(was_useful_collision({WILD, data->x, data->d}))
         {
            kangaroo_go_flag = false;
            found_useful_flag = true;
            result = (N/2 + uc_old.d - uc_new.d) % N;
         }
         lck.unlock();
```

```
      }

      std::this_thread::yield();
   }
}

cpp_int kangaroo()
{
   // Initialize the jump distances
   while(S.size() == 0 ||
        cpp_int(std::accumulate(S.begin(), S.end(), cpp_int(0)) / S.
           ↪ size())
         < mp::sqrt(cpp_int(U*V*(N-1)))/2)
   { // All step sizes are powers of 2, starting from 1
      S.emplace_back(U*V*mp::pow(cpp_int(2), S.size()));
      J.push_back(mp::powm(g, S.back(), N));
   }

   // Mean value and spacing measures
   cpp_int beta = cpp_int(std::accumulate(S.begin(), S.end(), cpp_int
      ↪ (0)) / S.size());
   cpp_int v = beta / (M/2);

   // Starting locations and distances
   for(size_t i = 0; i < U; ++i)
   { // Tame
      t0s[i].x = mp::powm(g, cpp_int((N-1)/2 + i*V), N);
      #ifdef DEBUG
      std::cout << cpp_int((N-1)/2 + i*V) << "␣->␣" << mp::powm(g,
         ↪ cpp_int((N-1)/2 + i*V), N) << std::endl;
      #endif
      t0s[i].d = 0;
   }
   for(size_t i = 0; i < V; ++i)
   { // Wild
      w0s[i].x = mp::powm(g, i*V, N);
      w0s[i].x *= (w0s[i].x * t) % N;
      w0s[i].d = 0;
      #ifdef DEBUG
      std::cout << i*V << "␣=>␣" << mp::powm(g, i*V, N) << "␣=>␣" <<
         ↪ w0s[i].x << std::endl;
      #endif
   }

   // Set the distinguished points mask
```

```
    #ifdef DPMASK_VALUE
    dpm = DPMASK_VALUE;
    #else
    dpm = 15;
    #endif

    // Make threads
    for(int i = 0; i < U; ++i) { threads[ i] = std::thread(tame_kangaroo
        ↪ , (void *)(&t0s[i])); }
    for(int i = 0; i < V; ++i) { threads[U+i] = std::thread(
        ↪ wild_kangaroo, (void *)(&w0s[i])); }

    // Launch the threads
    kangaroo_go_flag = true;

    // Wait for a collision
    for(auto &t : threads) { t.join(); }

    // Have one! We can return the answer now
    return result;
}

int main(int argc, char **argv)
{
    if (argc != 4)
    {
        std::cout << "Usage: ###.exe <Safe prime N> <Generator g> <Target
            ↪ t>" << std::endl;
        return 2;
    }

    // Log base <generator>
    g = cpp_int(argv[1]);
    t = cpp_int(argv[2]);
    N = cpp_int(argv[3]);

    std::stringstream ss;
    ss << "Result: " << kangaroo() << '\n';
    std::cout << ss.str() << std::flush;

    return 0;
}
```

E DLP Answer Verification Tool

```
#include <iostream>
#include <boost/multiprecision/cpp_int.hpp>

using boost::multiprecision::cpp_int;

void print_help() { std::cout << "Usage: dlp_verifier <g> <x> <p>" <<
    ↪ std::endl; }

int main(int argc, char **argv)
{
    if(argc != 4) { print_help(); return 0; }
    cpp_int g(argv[1]);
    cpp_int x(argv[2]);
    cpp_int p(argv[3]);
    std::cout << boost::multiprecision::powm(g, x, p) << std::endl;
    return 0;
}
```

F DLP Generation Tool

```
#include <chrono>
#include <iostream>
#include <fstream>
#include <random>
#include <sstream>
#include <boost/multiprecision/cpp_int.hpp>
#include <boost/multiprecision/miller_rabin.hpp>

using boost::multiprecision::cpp_int;
using boost::multiprecision::miller_rabin_test;
namespace mp = boost::multiprecision;

// Instantiate the random number generator
const unsigned seed = std::chrono::system_clock::now().time_since_epoch
   ↪ ().count();
std::mt19937_64 generator(seed);

unsigned ask_user_number()
{
   unsigned nbits = -1;
   std::cout << "How␣many␣bits␣should␣the␣prime␣be␣(enter␣a␣number␣on␣
      ↪ the␣interval␣[8,␣400])?␣";
   std::cin >> nbits;
   std::cin.ignore();
   if(nbits == -1) { std::cout << nbits << std::endl; return
      ↪ ask_user_number(); }
   return nbits;
}

int main(int argc, char **argv)
{
   short nbits, loops_without_success;
   int nloops = (argc > 2 ? atoi(argv[2]) : 0);

   if(argc > 1) { nbits = atoi(argv[1]); } // Bits provided, do not
      ↪ prompt
   else { nbits = ask_user_number(); } // Bits not provided, so prompt

   if(argc <= 2)
   { // Print if not going to file
      std::cout << "Generating␣DLP␣problems␣with␣primes␣of␣size␣" <<
         ↪ nbits
```

```
            << ". Press Enter to generate additional problems." <<
               ↪ std::endl;
   }

   // Checking numbers starting from the biggest possible nbits-bit
      ↪ number
   cpp_int x = mp::pow(cpp_int(2), nbits)-1;
   cpp_int x0 = x;

   cpp_int i = 0;
   while(x > 0)
   {
      // Hunting for a prime x...
      if(boost::multiprecision::miller_rabin_test(x, 40))
      { //...for which (x-1)/2 is also prime (i.e. x is a safe prime).
         if(boost::multiprecision::miller_rabin_test((x-1)/2, 40))
         { //...and for which x = 3 mod 8, which means 2 is a generator
            ↪ .
            if(x % 8 == 3)
            {
outputlabel:
               // Generate a random target number on [1, x).
               cpp_int target_number(0);
               for(short i = 0; i < 64; ++i)
               {
                  target_number = (target_number << 64) | generator();
                  target_number = target_number % x;
               }

               // Create this DLP instance's line
               // If 500 bits or more, output in hexadecimal
               std::stringstream ss("");
               ss << (nbits < 500 ? std::dec : std::hex)
                  << "2 ^ x = "
                  << (nbits < 500 ? "" : "0x")
                  << target_number
                  << " mod "
                  << (nbits < 500 ? "" : "0x")
                  << x;

               // Output to either stdout or file depending on mode:
               if(argc <= 2)
               { // Mode is stdout
                  std::cout << ss.str() << " ";
                  std::string repeat;
```

```
                std::getline(std::cin, repeat);
                if(!(repeat == "q" || repeat == "Q")) { goto
                    ↪ outputlabel; }
            }
            else if(nloops-- > 0)
            { // Mode is file
                std::fstream fs(argv[3], std::fstream::out | std::
                    ↪ fstream::app);
                fs << ss.str() << std::endl;
                fs.close();
                goto outputlabel;
            }

            return 0;
        }
    }
}

    // Failed to find anything, try next number down
    --x;
}

// No suitable primes were found at all. Return failure.
return 1;
}
```

G DLP Problems

G.1 8 Bit Prime Modulus

```
2 ^ x = 118 mod 227
2 ^ x = 1 mod 227
2 ^ x = 16 mod 227
2 ^ x = 164 mod 227
2 ^ x = 203 mod 227
2 ^ x = 180 mod 227
2 ^ x = 121 mod 227
2 ^ x = 215 mod 227
2 ^ x = 175 mod 227
2 ^ x = 80 mod 227
2 ^ x = 145 mod 227
2 ^ x = 6 mod 227
2 ^ x = 217 mod 227
2 ^ x = 20 mod 227
2 ^ x = 14 mod 227
2 ^ x = 20 mod 227
2 ^ x = 190 mod 227
2 ^ x = 219 mod 227
2 ^ x = 13 mod 227
2 ^ x = 153 mod 227
```

G.2 16 Bit Prime Modulus

```
2 ^ x = 51986 mod 65267
2 ^ x = 44757 mod 65267
2 ^ x = 36647 mod 65267
2 ^ x = 10314 mod 65267
2 ^ x = 39015 mod 65267
2 ^ x = 55679 mod 65267
2 ^ x = 50744 mod 65267
2 ^ x = 59257 mod 65267
2 ^ x = 47720 mod 65267
2 ^ x = 9306 mod 65267
2 ^ x = 18625 mod 65267
2 ^ x = 22898 mod 65267
2 ^ x = 5157 mod 65267
2 ^ x = 14853 mod 65267
2 ^ x = 40440 mod 65267
2 ^ x = 64846 mod 65267
```

```
2 ^ x = 46725 mod 65267
2 ^ x = 3537 mod 65267
2 ^ x = 59868 mod 65267
2 ^ x = 10540 mod 65267
```

G.3 32 Bit Prime Modulus

```
2 ^ x = 100338841 mod 4294963787
2 ^ x = 2790431031 mod 4294963787
2 ^ x = 3353293824 mod 4294963787
2 ^ x = 4270130900 mod 4294963787
2 ^ x = 2933198008 mod 4294963787
2 ^ x = 1340717910 mod 4294963787
2 ^ x = 1301686175 mod 4294963787
2 ^ x = 2011475840 mod 4294963787
2 ^ x = 3395228340 mod 4294963787
2 ^ x = 2646590003 mod 4294963787
2 ^ x = 4202600729 mod 4294963787
2 ^ x = 1338932882 mod 4294963787
2 ^ x = 3925491979 mod 4294963787
2 ^ x = 1050895730 mod 4294963787
2 ^ x = 3936909613 mod 4294963787
2 ^ x = 3305250565 mod 4294963787
2 ^ x = 621534413 mod 4294963787
2 ^ x = 1557635495 mod 4294963787
2 ^ x = 195391631 mod 4294963787
2 ^ x = 2755418519 mod 4294963787
```

G.4 34 Bit Prime Modulus

```
2 ^ x = 9604710549 mod 17179867547
2 ^ x = 7753536673 mod 17179867547
2 ^ x = 7293896668 mod 17179867547
2 ^ x = 776539681 mod 17179867547
2 ^ x = 11264868005 mod 17179867547
2 ^ x = 1027172925 mod 17179867547
2 ^ x = 3411475899 mod 17179867547
2 ^ x = 6260974714 mod 17179867547
2 ^ x = 9574550003 mod 17179867547
2 ^ x = 3036538151 mod 17179867547
2 ^ x = 2830682710 mod 17179867547
2 ^ x = 9397403214 mod 17179867547
2 ^ x = 10010475028 mod 17179867547
```

```
2 ^ x = 3236106084 mod 17179867547
2 ^ x = 12122343623 mod 17179867547
2 ^ x = 3400573249 mod 17179867547
2 ^ x = 12040021937 mod 17179867547
2 ^ x = 11039724443 mod 17179867547
2 ^ x = 13441108362 mod 17179867547
2 ^ x = 16950892619 mod 17179867547
```

G.5 36 Bit Prime Modulus

```
2 ^ x = 4133996098 mod 68719474427
2 ^ x = 45828031702 mod 68719474427
2 ^ x = 66049221687 mod 68719474427
2 ^ x = 50576552704 mod 68719474427
2 ^ x = 63536488161 mod 68719474427
2 ^ x = 1940654348 mod 68719474427
2 ^ x = 17319359343 mod 68719474427
2 ^ x = 66392138817 mod 68719474427
2 ^ x = 17741809276 mod 68719474427
2 ^ x = 10419807584 mod 68719474427
2 ^ x = 47660198973 mod 68719474427
2 ^ x = 19333931897 mod 68719474427
2 ^ x = 63898805905 mod 68719474427
2 ^ x = 46588259032 mod 68719474427
2 ^ x = 6697991015 mod 68719474427
2 ^ x = 35859784817 mod 68719474427
2 ^ x = 66407337280 mod 68719474427
2 ^ x = 15553393300 mod 68719474427
2 ^ x = 64638160766 mod 68719474427
2 ^ x = 49965755724 mod 68719474427
```

G.6 38 Bit Prime Modulus

```
2 ^ x = 231056761065 mod 274877906243
2 ^ x = 203802858100 mod 274877906243
2 ^ x = 62606892149 mod 274877906243
2 ^ x = 6129730535 mod 274877906243
2 ^ x = 250073408131 mod 274877906243
2 ^ x = 10035322984 mod 274877906243
2 ^ x = 193668293232 mod 274877906243
2 ^ x = 111177741569 mod 274877906243
2 ^ x = 147450972680 mod 274877906243
2 ^ x = 109552031966 mod 274877906243
```

```
2 ^ x = 54418582633 mod 274877906243
2 ^ x = 268420976792 mod 274877906243
2 ^ x = 217120704111 mod 274877906243
2 ^ x = 73728084086 mod 274877906243
2 ^ x = 246648873203 mod 274877906243
2 ^ x = 81027263783 mod 274877906243
2 ^ x = 175691574322 mod 274877906243
2 ^ x = 131197416498 mod 274877906243
2 ^ x = 62988542992 mod 274877906243
2 ^ x = 160689967334 mod 274877906243
```

G.7 40 Bit Prime Modulus

```
2 ^ x = 1007345823806 mod 1099511627339
2 ^ x = 778905426082 mod 1099511627339
2 ^ x = 406429697112 mod 1099511627339
2 ^ x = 853357517656 mod 1099511627339
2 ^ x = 117810356988 mod 1099511627339
2 ^ x = 894644852124 mod 1099511627339
2 ^ x = 626301015639 mod 1099511627339
2 ^ x = 98025513939 mod 1099511627339
2 ^ x = 305026518442 mod 1099511627339
2 ^ x = 98538111657 mod 1099511627339
2 ^ x = 773805039475 mod 1099511627339
2 ^ x = 827398884008 mod 1099511627339
2 ^ x = 77610082015 mod 1099511627339
2 ^ x = 29529473912 mod 1099511627339
2 ^ x = 530209530809 mod 1099511627339
2 ^ x = 70963802888 mod 1099511627339
2 ^ x = 51153265514 mod 1099511627339
2 ^ x = 810291493051 mod 1099511627339
2 ^ x = 387431307924 mod 1099511627339
2 ^ x = 813237463151 mod 1099511627339
```

G.8 48 Bit Prime Modulus

```
2 ^ x = 18493327237936 mod 281474976704939
2 ^ x = 223435953638238 mod 281474976704939
2 ^ x = 80287252440143 mod 281474976704939
2 ^ x = 97820495029190 mod 281474976704939
2 ^ x = 193345929076799 mod 281474976704939
2 ^ x = 174873991936382 mod 281474976704939
2 ^ x = 36598180315902 mod 281474976704939
```

```
2 ^ x = 137988440487350 mod 281474976704939
2 ^ x = 8103506316113 mod 281474976704939
2 ^ x = 159813571106910 mod 281474976704939
2 ^ x = 166027617408862 mod 281474976704939
2 ^ x = 125692912480089 mod 281474976704939
2 ^ x = 214394844514393 mod 281474976704939
2 ^ x = 41433017574403 mod 281474976704939
2 ^ x = 223099247717126 mod 281474976704939
2 ^ x = 172931037645148 mod 281474976704939
2 ^ x = 152902684958685 mod 281474976704939
2 ^ x = 44034959166655 mod 281474976704939
2 ^ x = 211831674368415 mod 281474976704939
2 ^ x = 121122475261262 mod 281474976704939
```

G.9 56 Bit Prime Modulus

```
2 ^ x = 863946575168767 mod 72057594037925099
2 ^ x = 56385344601640814 mod 72057594037925099
2 ^ x = 2709079066753259 mod 72057594037925099
2 ^ x = 12003687111140376 mod 72057594037925099
2 ^ x = 17816190856259531 mod 72057594037925099
2 ^ x = 10861179441322317 mod 72057594037925099
2 ^ x = 65164534343682967 mod 72057594037925099
2 ^ x = 20439890256125015 mod 72057594037925099
2 ^ x = 19864996085884413 mod 72057594037925099
2 ^ x = 63298527616596506 mod 72057594037925099
2 ^ x = 10383570282503676 mod 72057594037925099
2 ^ x = 11817976067679321 mod 72057594037925099
2 ^ x = 11329909696939296 mod 72057594037925099
2 ^ x = 43357682710043405 mod 72057594037925099
2 ^ x = 19497614677333632 mod 72057594037925099
2 ^ x = 67854027025964233 mod 72057594037925099
2 ^ x = 39766222271739346 mod 72057594037925099
2 ^ x = 70146300877059103 mod 72057594037925099
2 ^ x = 55058823701112518 mod 72057594037925099
2 ^ x = 57700773652252932 mod 72057594037925099
```

G.10 64 Bit Prime Modulus

```
2 ^ x = 16534071812745773567 mod 18446744073709550147
2 ^ x = 15697062545178016561 mod 18446744073709550147
2 ^ x = 4693669573443213493 mod 18446744073709550147
2 ^ x = 5506789009040414064 mod 18446744073709550147
```

```
2 ^ x = 13396810419630237931 mod 18446744073709550147
2 ^ x = 2622953218250771698 mod 18446744073709550147
2 ^ x = 14132220759098468622 mod 18446744073709550147
2 ^ x = 18151840078348499753 mod 18446744073709550147
2 ^ x = 10579699359381281633 mod 18446744073709550147
2 ^ x = 18204284001518663117 mod 18446744073709550147
2 ^ x = 6795583635438079571 mod 18446744073709550147
2 ^ x = 5018600480340782125 mod 18446744073709550147
2 ^ x = 14352161994976539317 mod 18446744073709550147
2 ^ x = 663349713174927861 mod 18446744073709550147
2 ^ x = 15136250537522405077 mod 18446744073709550147
2 ^ x = 8943665046688276252 mod 18446744073709550147
2 ^ x = 2373532040798395246 mod 18446744073709550147
2 ^ x = 4728339324710090537 mod 18446744073709550147
2 ^ x = 4948316605888183536 mod 18446744073709550147
2 ^ x = 14539943888572184598 mod 18446744073709550147
```

G.11 72 Bit Prime Modulus

```
2 ^ x = 4562423733255227245266 mod 4722366482869645207163
2 ^ x = 4707618002935753839791 mod 4722366482869645207163
2 ^ x = 2263907409657962038812 mod 4722366482869645207163
2 ^ x = 2238468377868763382288 mod 4722366482869645207163
2 ^ x = 3594174012847121644850 mod 4722366482869645207163
2 ^ x = 3450284233133644033817 mod 4722366482869645207163
2 ^ x = 1343523096234987941561 mod 4722366482869645207163
2 ^ x = 3035941022383454915728 mod 4722366482869645207163
2 ^ x = 4545435940127118024126 mod 4722366482869645207163
2 ^ x = 268677153607086325556 mod 4722366482869645207163
2 ^ x = 1310468543512057833527 mod 4722366482869645207163
2 ^ x = 338862042442409403804 mod 4722366482869645207163
2 ^ x = 3107545102346137060536 mod 4722366482869645207163
2 ^ x = 738623484622727750704 mod 4722366482869645207163
2 ^ x = 1181130635212186886299 mod 4722366482869645207163
2 ^ x = 183919283056668244923 mod 4722366482869645207163
2 ^ x = 158283029578374602417 7 mod 4722366482869645207163
2 ^ x = 2581712082643689880802 mod 4722366482869645207163
2 ^ x = 956772066306334488309 mod 4722366482869645207163
2 ^ x = 940233611842023464867 mod 4722366482869645207163
```

G.12 80 Bit Prime Modulus

```
2 ^ x = 685495437812517396612292 mod 1208925819614629174700339
```

2 ^ x = 267271362226332484306975 mod 1208925819614629174700339
2 ^ x = 168607093398848041499227 mod 1208925819614629174700339
2 ^ x = 364724138953697279539901 mod 1208925819614629174700339
2 ^ x = 543259629369539311467180 mod 1208925819614629174700339
2 ^ x = 697356764069089492721985 mod 1208925819614629174700339
2 ^ x = 250112773792725640824337 mod 1208925819614629174700339
2 ^ x = 304966713401886581266 26 mod 1208925819614629174700339
2 ^ x = 672633704754092037700769 mod 1208925819614629174700339
2 ^ x = 1133409974524849799384827 mod 1208925819614629174700339
2 ^ x = 534546982747070065329584 mod 1208925819614629174700339
2 ^ x = 1183872110657586786025811 mod 1208925819614629174700339
2 ^ x = 860850608538967161460189 mod 1208925819614629174700339
2 ^ x = 223963739255907586328939 mod 1208925819614629174700339
2 ^ x = 723874004053407772188752 mod 1208925819614629174700339
2 ^ x = 467720908499300879425462 mod 1208925819614629174700339
2 ^ x = 757540198642050757553184 mod 1208925819614629174700339
2 ^ x = 106313546976406044285907 4 mod 1208925819614629174700339
2 ^ x = 583966446743928446159 70 mod 1208925819614629174700339
2 ^ x = 118977615022101804608255 6 mod 1208925819614629174700339