COMPARISON OF RANDOM SQUARES AND INDEX CALCULUS ALGORITHMS FOR INTEGER FACTORIZATION AND DISCRETE LOGARITHMS

YIMENG FANG

Abstract. This project compared the Random Squares algorithm for integer factorization and the Index Calculus algorithm for discrete logarithms. Both methods transform multiplicative problems into linear relations using factor bases and randomly generated candidates. Controlled experiments in SageMath tested runtime performance across 12-22-bit integer sizes. Results show a crossover at approximately 16-bit integers: Index Calculus is faster for smaller integers, while Random Squares outperforms for larger integers with more stable performance. These findings highlight the differing computational strategies and practical efficiency of the two algorithms.

1. Introduction

The security of modern cryptographic systems does not rely on the secrecy of the algorithms themselves. Instead, it is based on the assumption that certain computational problems are impossible to solve within a reasonable time. It means that operations like encryption and decryption can be performed efficiently by users in a rather short time, while for an attacker, recovering the private key from public information is considered computationally impossible. For RSA, its security relies on the difficulty of large integer factorization, while for elliptic curve cryptography (ECC), it relies on the difficulty of the discrete logarithm problem (DLP). Although various theoretical methods exist, in real world applications, the computational efficiency of different algorithms varies significantly.

Among classical algorithms targeting large integer N factorization and the discrete logarithm problem in the multiplicative group $(\mathbb{Z}/N\mathbb{Z})^\times$, there are two methods that follow a similar fundamental approach, the random square algorithm, also known as the quadratic sieve method [4, Section 5.5], and the index calculus algorithm [3, Section 3.8]. These algorithms share a common structure: they search for special congruences or dependency relations and represent these relations as factorization forms relative to a fixed set of small prime numbers (the factor base). By continuously generating and filtering these relations, the algorithm collects a sufficient number of "survivors", allowing subsequent steps to extract information using linear algebra techniques. Consequently, in practical applications, the computational efficiency of these algorithms varies a lot. This efficiency is heavily influenced by key parameters, such as the size of the factor base, the number of survivors, and, most importantly, the bit size of the integer involved.

Based on this, this project primarily focuses on the following question: For integer factorization and the discrete logarithm problem, across composite integers N of varying bit sizes (12 - 22 bits), at which bit sizes does the random squares algorithm show superior computational efficiency compared

Date: January 6, 2026.
This Honours Project at the University of Ottawa was conducted under the supervision of Monica Nevins.

to the index calculus algorithm, under the constraint of identical factor base and equal number of survivors.

1.1. **Summary of Results.** The experimental results indicate that a clear crossover in efficiency between the two algorithms occurs when the bit size is approximately 16. For integers smaller than 16 bits, the index calculus method shows better performance. For integers larger than 16 bits, the random squares method becomes more efficient. Furthermore, the experiments analyzed key factors influencing algorithm performance, including the size of the factor base and the number of survivors, the composition of the factor base, and the maximum number of attempts for finding survivors.

1.2. **Limitations and Future Research Directions.** Although the experimental results provide a valid comparison, some limitations remain. Firstly, due to constraints in experimental conditions and parameters, this study primarily analyzed the efficiency of successful factorization instances. It did not investigate the overall success probability for larger integers. Future research could explore the performance and success rate variations of these algorithms under larger bit sizes and different parameter configurations.

2. Background

2.1. **Cryptography.** This section provides the necessary cryptographic background for the algorithms studied in this project. We briefly introduce the basic principles of modern public-key cryptography, focusing on RSA and elliptic curve cryptography (ECC), and highlight the mathematical problems underlying their security, namely integer factorization and the discrete logarithm problem. These concepts will serve as the foundation for the algorithmic discussions and experimental comparisons in the subsequent sections. Our basic references for the material in this section are the course notes [1] and the textbook [4].

2.1.1. *Introduction to Modern Cryptography.* Cryptography enables secure communication between users without the need to share secret information in advance. Each user holds a pair of keys: a public key and a private key. The public key is used to encrypt messages, while the private key is used to decrypt them. The core functionality of a public key cryptosystem can be summarized in three steps:

(1) Key Generation: Generate a public key and a private key.
(2) Encryption: Transform a plaintext message into ciphertext using the public key.
(3) Decryption: Recover the original plaintext from the ciphertext using the private key.

The security of such a mechanism does not rely on keeping the algorithm secret. Instead, it depends on the computational difficulty of certain mathematical problems. Only a user in possession of the correct key can perform encryption or decryption operations efficiently. For an unauthorized attacker, even with full knowledge of the algorithm, decrypting the information without the key is infeasible within any reasonable time. RSA and ECC are classic examples of public key cryptosystems based on mathematically hard problems. Their security relies on the Integer Factorization Problem and the Discrete Logarithm Problem, respectively.

2.1.2. *RSA.* RSA is one of the earliest and most widely used public-key cryptosystems. The basic workflow is as follows:

(1) Key Generation:
   (a) Choose two distinct large prime numbers, p and q, typically with sufficiently large bit-lengths to ensure security.
   (b) Compute their product $N = p \cdot q$
   (c) Compute $\varphi(N) = (p-1)(q-1)$.
   (d) Choose a public key e such that $\gcd(e, \varphi(N)) = 1$.
   (e) Compute the private key d such that $d \cdot e \equiv 1 \pmod{\varphi(N)}$.

   The public key is the pair (N, e), and the private key is the pair (N, d).
(2) Encryption: Given a plaintext message m with $0 \leq m < N$, compute the ciphertext: $c \equiv m^e \pmod{N}$.
(3) Decryption: After receiving the ciphertext c, use the private key d to recover the plaintext: $m' \equiv c^d \pmod{N}$.

The correctness is ensured because $m' \equiv c^d \equiv (m^e)^d \equiv m^{e \cdot d} \equiv m^{1+k \cdot \varphi(N)} \equiv m \pmod{N}$. To break RSA, it is widely believed that one must compute p and q to derive $\varphi(N)$ and subsequently the private key d. Therefore, the security of RSA is directly dependent on the difficulty of factoring large integers: the ability to factor N quickly would break RSA. In the absence of efficient factoring algorithms, RSA is considered secure. As a result, we can say the security of RSA is based on the Integer Factorization Problem: given a composite integer N, determine its nontrivial prime factors p and q.

2.1.3. *ECC.* ECC is another public-key cryptosystem based on a mathematically hard problem. Its security relies on the Elliptic Curve Discrete Logarithm Problem (ECDLP). The basic workflow is as follows:

(1) Key Generation:
   (a) Select an elliptic curve E defined over a finite field $\mathbb{F}_p$.
   (b) Select a base point $P \in E(\mathbb{F}_p)$ to serve as a group generator.
   (c) A user randomly selects an integer b as the private key.
   (d) Compute the public key $Q = b \cdot P$, using scalar multiplication on the elliptic curve.
(2) Encryption:
   (a) Given a plaintext message m, randomly select an integer a, then encoded m as a point P_m on the curve
   (b) Generate the ciphertext pair: $(R, S) = (a \cdot P,\ P_m + a \cdot Q)$.
(3) Decryption:
   (a) Recover the message point by computing: $S - b \cdot R = P_m$.

The correctness is ensured because $Q = b \cdot P$, so $S - b \cdot R = P_m + a \cdot (b \cdot P) - b \cdot (a \cdot P) = P_m$ and the elliptic curve group is abelian. The security of ECC is based on the Discrete Logarithm Problem (DLP). In the ECC, the private key is an integer b, and the corresponding public key is the point $Q = bP$, where P is a publicly known base point on the curve. Given P and Q, the problem of recovering the integer b is known as the elliptic curve discrete logarithm problem. More generally, given a cyclic group G with generator g, the Discrete Logarithm Problem (DLP) is the problem of, given an element $h \in G$, determining an integer a such that $g^a = h$. No efficient classical algorithm is known for solving the DLP in general groups of cryptographic size, and the presumed hardness of this problem underlies the security of ECC.

2.1.4. *Classical and Quantum Algorithms for Factoring and Discrete Logarithms.* In the above sections, we introduced that the security of RSA relies on the integer factorization problem, while the security of ECC relies on the discrete logarithm problem. To understand the computational complexity of these mathematically hard problems, scientists have proposed various algorithms for solving integer factorization or discrete logarithm problems. In this section, we will briefly summarize a few representative algorithms.

(1) Integer Factorization Algorithms:
   (a) Fermat's Factorization Method: The main idea is based on a fundamental lemma: for an odd integer m that is not a perfect square, factoring $m = ab$ is equivalent to representing m as the difference of two squares, i.e., $m = t^2 - s^2 = (t + s)(t - s)$. Conversely, if $m = ab$ is known, we can set $t = \frac{a+b}{2}$ and $s = \frac{|a-b|}{2}$, and again obtain $m = t^2 - s^2$. Therefore, the problem of integer factorization can be reduced to the task of finding a representation of the number as a difference of two squares. In practice, perfect squares are relatively rare. Hence, if s^2 is small, we can assume that $t \approx \lceil \sqrt{m} \rceil$. Then, we can start from $t = \lceil \sqrt{m} \rceil$ and find a value of t such that $t^2 - m$ is a perfect square, and we may quickly find the factorization.
   (b) Shor's Algorithm: Shor's algorithm is a quantum algorithm for factoring integers. The main idea is to find a r such that $a^r \equiv 1 \pmod{N}$. If r is even and $a^{r/2} \not\equiv -1 \pmod{N}$, then a factor of N can be found by computing $\gcd(a^{r/2} \pm 1, N)$. If these conditions are not met, the algorithm is repeated with a different a. The quantum part of the algorithm efficiently computes the order r, which allows it to solve the factoring problem in polynomial time. Although current quantum computers are not yet large enough to break RSA, this algorithm shows that integer factorization problem is easy in theory on a quantum computer.
   (c) Random Squares Algorithm: The Random Squares algorithm is a classical integer factorization method. Its core idea is to find integers x and y such that $x^2 \equiv y^2 \pmod{N}$, and then extract a non-trivial factor by calculating $\gcd(x - y, N)$ This algorithm is the subject of detailed discussion in Section 2.2 and serves as one of the core algorithms compared in our experiments.
(2) Discrete Logarithm Algorithms: The discrete logarithm problem (DLP) in its general form considers a cyclic group G with a generator g and an element h. The goal is to find an integer a such that $g^a = h$.
   (a) Baby-step Giant-step Algorithm: The core idea of the algorithm is to split the exponent a into two components, $a = k + \ell \cdot x$, where $x \approx \sqrt{N}$ (N is the order of the group) and $1 \leq k, \ell \leq x$. Using group operations, the discrete logarithm equation can be rewritten as $h = g^a = g^{k+\ell x} = g^k \cdot (g^x)^\ell$.
   The algorithm proceeds by constructing two tables: one storing the precomputed values of g^k for all k (the "baby steps"), and another storing the computed values of $h \cdot (g^x)^{-\ell}$ for each ℓ (the "giant steps"). When a match is found between the tables—that is, when $g^k = h \cdot (g^x)^{-\ell}$ the exponent can be recovered as $a = k + \ell x$. This approach decomposes the DLP into a smaller search-and-match problem by precomputing and storing a subset of values.
   (b) Pohlig–Hellman Algorithm: The Pohlig–Hellman algorithm uses the prime factorization of the group order to break down a discrete logarithm problem into several smaller problems. Suppose the order N of the group can be factored into prime powers, $N = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$. For each prime power $p_i^{e_i}, h = g^a$ can be projected onto the subgroup of order $p_i^{e_i}$. Solving the equation in this subgroup, we can have a partial solution $a_i \bmod p_i^{e_i}$. Then, by combining these partial solutions using the Chinese Remainder

Theorem, we obtain the solution a to the original problem. The main idea of this method is to decompose the DLP of a large group into the DLP of multiple smaller groups.

(c) Index Calculus Algorithm: The Index Calculus Algorithm is a class of linear algebra-based algorithms for solving discrete logarithms. This method only applies to the multiplicative group $(\mathbb{Z}/N\mathbb{Z})^\times$, as it relies on arithmetic properties of the integers, in particular primality. The main idea is to pre-selecting a set of small primes as a factor base, finding numbers that factor completely over this base. These numbers are called survivors. Once enough survivors are collected, we can then construct a system of linear equations to solve for the discrete logarithm. This method is similar to the Random Squares algorithm. We will describe it in detail in Section 2.3.

2.2. **The Random Squares Method.** We now introduce the Random Squares Algorithm in detail.

2.2.1. *Theoretical Background.* The Random Squares Algorithm generalizes the key idea of Fermat factorization: finding a pair of integers x and y such that $x^2 \equiv y^2 \pmod N$ and $x \not\equiv \pm y \pmod N$. Once such a pair is found, the greatest common divisor $\gcd(x+y, N)$ is a nontrivial factor of N with probability approximately $1/2$. This is because for a composite $N = pq$ with distinct primes p and q, the congruence $x^2 \equiv y^2 \pmod N$ implies $x^2 \equiv y^2 \pmod p$ and $x^2 \equiv y^2 \pmod q$. Each of these congruences has two solutions, giving a total of four combinations modulo N, half of which satisfy $x \not\equiv \pm y \pmod N$ and yield a nontrivial factor.

To systematically construct such square congruences, the Random Squares algorithm introduces the concepts of a factor base and smooth numbers. A factor base FB is typically defined as a finite set of the smallest primes, for example $FB = \{2, 3, 5, \ldots, p_t\}$, where t is chosen according to the size of N. Additionally, -1 can be included in the factor base to account for sign changes in factorizations. In the following context, we refer to the elements of the factor base as primes, even if the set includes -1, for convenience.

2.2.2. *Algorithm Overview.* The algorithm proceeds through five main steps:

(1) Step 1: Factor Base Selection: Choose a factor base $FB = \{-1, 2, 3, \ldots, p_t\}$, a small set of primes. The factor base is used to find numbers that factor completely over FB, called survivors. An integer $y = x^2 \bmod N$ is considered a survivor if it can be factored over FB.
(2) Step 2: Random Candidate Generation: Repeatedly pick random integers x with $0 < x < N$ and compute $y = x^2 \bmod N$. Each y is then tested to see if it can be factored entirely over the factor base.
(3) Step 3: Survivor Selection: If y can be factored over FB, record its factorization as a vector over $\mathbb{Z}_2$: $\alpha_i = (a_{i,0}, a_{i,1}, \ldots, a_{i,t}) \in \mathbb{Z}_2^{t+1}$, where $a_{i,j}$ is the exponent of the prime p_j in y_i modulo 2, and $a_{i,0}$ corresponds to -1 if included. Such y_i is called a survivor. Repeat this process until the number of survivors m exceeds $t+1$. Collecting enough survivors is typically the most time-consuming step.
(4) Step 4: Finding Linear Dependencies: Consider the m vectors α_i over $\mathbb{Z}_2$. Since $m > t+1$, these vectors are linearly dependent. A dependency corresponds to a subset $J \subset \{1, \ldots, m\}$ such that $\sum_{j\in J} \alpha_j \equiv 0 \pmod 2$. This implies that the product of the corresponding y_j is a perfect square: $\prod_{j\in J} y_j = \prod_{j\in J} (-1)^{a_{j,0}} p_1^{a_{j,1}} \cdots p_t^{a_{j,t}} = Y^2$. At the same time, the product of the corresponding x_j gives $X = \prod_{j\in J} x_j$, so that $X^2 \equiv Y^2 \pmod N$.

(5) Step 5: Factor Extraction: Once a pair X, Y with $X^2 \equiv Y^2 \pmod{N}$ is found, calculate $\gcd(X+Y, N)$ and $\gcd(X-Y, N)$ to obtain a nontrivial factor of N. With probability approximately $1/2$, one of these gcds gives a proper factor. If a factor is not found, select a different dependency (if multiple exist) or continue generating more survivors until a new linear dependency is found.

The efficiency of this algorithm is primarily due to two factors: first, the probability of finding survivors, which determines the speed of step 3; and second, the computational cost of solving the linear system modulo 2, which is step 4. A main characteristic of the Random Squares algorithm is the randomness in its relation generation process. In comparison, the Index Calculus Algorithm also involves some randomness in candidate selection, but combines this with a structured sieving approach.

2.3. **The index Calculus Method.** We now introduce the index Calculus Algorithm in detail.

2.3.1. *Theoretical Background.* The index Calculus method is a structured approach to solving the DLP in a chosen cyclic group. The key idea is to fix a factor base and express other group elements in terms of this factor base. By doing so, the original hard problem is reduced to solving a system of linear equations over a finite ring.

To facilitate a fair comparison with the Random Squares Algorithm, we consider the DLP in the multiplicative group $G = \mathbb{Z}_p^*$ of a prime p. In the context of the DLP in $\mathbb{Z}_p^*$ for a prime p, let g be a generator of the group. Choose a factor base $FB = \{p_1, p_2, \ldots, p_t\}$ consisting of the first t prime numbers. The algorithm proceeds in two phases: a precomputation phase, in which the discrete logarithms($\log_g(p_j)$) of the factor base elements are determined, and a computing phase, in which the $\log_g(p_j)$ of the target element $a \in \mathbb{Z}_p^*$ is computed using this information.

2.3.2. *Algorithm Overview.*

(1) Precomputation Phase.
   (a) Fix a generator g of the group $\mathbb{Z}_p^*$.
   (b) Choose a factor base $FB = \{p_1, p_2, \ldots, p_t\}$ consisting of the first t prime numbers. The size t is chosen to balance two competing effects:
      (i) a larger factor base increases the probability that random elements factor completely over FB;
      (ii) a smaller factor base reduces the cost of solving the resulting linear system.
   (c) Select integers a_i uniformly at random with $1 \leq a_i < p$, and compute $r_i \equiv g^{a_i} \pmod{p}$.
   (d) If r_i factors completely over the factor base, namely $r_i \equiv \prod_{j=1}^{t} p_j^{e_{i,j}} \pmod{p}$, then r_i is called a *survivor*. Only survivors are retained.
   (e) Take discrete logarithms (base g) of both sides to obtain the linear relation $a_i \equiv \sum_{j=1}^{t} e_{i,j} \log_g(p_j) \pmod{p-1}$.
   (f) Collect sufficiently many linearly independent relations and solve the resulting system of linear equations over $\mathbb{Z}_{p-1}$ to determine $\log_g(p_j)$ for all $p_j \in FB$.
(2) Computation Phase.
   (a) Given a target element $h \in \mathbb{Z}_p^*$, select random integers a until $g^a h \pmod{p}$ factors completely over the factor base.
   (b) Write $g^a h \equiv \prod_{j=1}^{t} p_j^{f_j} \pmod{p}$.
   (c) Take discrete logarithms to obtain $a + \log_g(h) \equiv \sum_{j=1}^{t} f_j \log_g(p_j) \pmod{p-1}$.

(d) Compute $\log_g(h)$ using the previously determined values of $\log_g(p_j)$.

2.4. **Connecting Factorization and the Discrete Logarithm Problem.** At a conceptual level, integer factorization and the discrete logarithm problem (DLP) are distinct computational problems. However, many of the most effective classical algorithms for both problems rely on the same principle: transforming multiplicative relations into linear relations by exploring special congruences. Both problems use a similar idea: to find nontrivial multiplicative relations using the prime factors of the modulus. In integer factorization, such relations are congruences, while in the DLP, they correspond to linear dependencies in exponent space. In both cases, these relations transform a difficult multiplicative problem into a simpler algebraic problem. This conceptual similarity forms the basis for comparing the performance of the Random Square method and the Index Calculus method under controlled conditions.

2.4.1. *Similarities Between the Two Methods.* The Random Squares method and the index Calculus method are built on this shared idea. In the Random Squares method, the goal is to find integers X and Y such that $X^2 \equiv Y^2 \pmod{N}$, then the factor of N would be a gcd. The algorithm achieves this by collecting survivors whose factorizations over the factor base correspond to linear dependencies.

Similarly, the index Calculus method seeks relations of the form $g^a \equiv \prod_j p_j^{e_j} \pmod{p}$, where the right-hand side factors over the factor base. These relations are converted into linear equations involving log_gs, which can then be solved using linear algebra.

In both cases, a factor base is fixed in advance, random elements are generated until they can be factored completely over this base (survivors), and the survivors are translated into linear systems. Thus, the index Calculus method can be viewed as a structured and specialized implementation of the same principle underlying the Random Squares method.

From an asymptotic perspective, both the Random Squares algorithm and the Index Calculus method have comparable running times. The Random Squares algorithm has subexponential complexity, which can be expressed as $O(\exp(2(\log N)^{1/2}(\log\log N)^{1/2}))$, often denoted by $O(L_N(1/2, 2))$, where $L_N(r, c) = \exp(c(\log N)^r(\log\log N)^{1-r})$ is a standard shorthand for subexponential functions [4, Section 5.5]. Similarly, the factor base method for solving the discrete logarithm problem in $\mathbb{Z}_p^*$ has complexity approximately $O(L_p(1/2, 2))$, which is also subexponential [3, Section 3.8]. This shows that, despite being applied to different problems, the two algorithms share the same asymptotic complexity.

2.4.2. *Motivation for Experimental Comparison.* Because both methods rely on the probability that randomly generated values factor completely over a chosen factor base (or equivalently, that a randomly generated value is a survivor), their performance is determined by similar assumptions. As a result, although the two algorithms are applied to different problems and operate in different settings, their behavior is expected to be quite similar. This theoretical similarity motivates a direct comparison of their performance under controlled conditions.

Given the close conceptual and structural relationship between the Random Squares method and the index Calculus method, it is natural to ask how their practical efficiency compares when key parameters are fixed. We therefore design experiments that control for:

(1) the composition of the factor base;

(2) the size of the factor base and the number of survivors collected;
(3) the number of attempts made to find survivors.

By comparing runtime under identical constraints, we aim to determine in which integer bit size one method outperforms the other, and whether a clear crossover point exists.

3. Experimental Design and Methodology

We implemented the two algorithms, Random Square Algorithm and Index Calculus Algorithm, in SageMath using CoCalc [2]. We include the code for the Index Calculus algorithm in Appendix B, and the code for the Random Square algorithm in Appendix A.

3.1. **Control Variables.**

To systematically compare the performance of the Index Calculus and Random Square factorization algorithms, we designed a series of controlled experiments. We selected 12 bits for parameter studies. Each parameter setting was run 50 times, and the average runtime was recorded for analysis.

Before discussing futher detail, we briefly describe the main steps of each algorithm. In the Index Calculus algorithm (see Listings B.2 and B.3 in Appendix B), for each randomly chosen exponent a, the algorithm computes $g^a \bmod p$ and attempts to factor the result over the fixed factor base. Similarly, in the Random Square algorithm (see Listings A.2 and A.3 in Appendix A), for each selected x, it computes $y = x^2 \bmod N$ and attempts to factor y completely over the factor base.

3.1.1. *Composition of factor base:*

A factor base is a set of prime numbers that used to test whether a candidate value is a survivor. Including -1 in the factor base would expand the search space to allow negative survivors, theoretically increasing the probability of finding valid relations. However, this also requires handling the decomposition of negative numbers and increases the computational cost for relation construction.

We tested two settings of the factor base: one including -1 and the other excluding it. In both cases, 10 survivors were collected per trial, and each setting was repeated 50 times. The results are summarized in Table 1. It indicates that excluding -1 from the factor base is a better choice for both methods. Therefore, to maximize overall efficiency and provide a streamlined framework for both algorithms, we decide to exclude -1 from the factor base in all subsequent experiments.

Table 1. Effect of including -1 in the factor base on average runtime and success rate.

Algorithm	Average Time (s)
Index Calculus with -1	0.001965
Index Calculus without -1	0.001559
Random Square with -1	0.002171
Random Square without -1	0.001846

3.1.2. *Number of survivors and size of factor base:*

To investigate the impact of the factor base size and the number of survivors on algorithm efficiency, we conducted controlled experiments varying both parameters simultaneously. For the Random Square method, increasing the factor base size requires more primes to be checked within the `factor_y` function (see Listing A.2), which increases the time for a single factorization. For the Index Calculus method, a larger factor base increases the

computational cost of checking whether a random choosen number can be factored over the factor base or is a survivor, as in the `setup_index_calculus` function (see Listing B.2).As the factor base grows, each number requires more time to check that it can be factored over the factor base.

We conducted experiments using two settings: a factor base size of 7 with 10 survivors (7,10), and a factor base size of 8 with 20 survivors(8,20). Each setting was repeated 50 times. The results are summarized in Table 2: increasing the factor base size from 7 to 8 and the number of survivors from 10 to 20 leads to higher runtimes for both algorithms. Although one might expect that a larger factor base and more survivors would improve performance by providing more factorization opportunities, the results indicate the opposite: the larger combination (8, 20) is less efficient. This is likely because the additional computational cost per candidate outweighs the benefit of having more available factors. Based on these findings, we selected the (7, 10) combination as the standard parameter setting for all subsequent experiments.

TABLE 2. Effect of factor base size and number of survivors on runtime and success rate (without -1).

Algorithm	Factor Base Size	Survivors	Average Time (s)
Index Calculus	7	10	0.001559
Index Calculus	8	20	0.006744
Random Square	7	10	0.001846
Random Square	8	20	0.002224

3.1.3. *Maximum attempts:* The maximum attempts parameter controls the number of candidate values the algorithm will try when searching for a valid factorization. Specifically, for the Random Square method, the algorithm randomly selects an integer x and computes $y = x^2 \bmod N$, checking if y can be fully factored over the factor base (see `factor_y` in Listing A.2). For the Index Calculus method, the algorithm randomly selects an exponent a and computes $s = g^a h \bmod p$, checking if s can be fully factored over the factor base (see `setup_index_calculus` in Listing B.2). The maximum attempts parameter sets an upper limit on how many y and s the algorithm will test.

We tested two settings for maximum attempts, 10^4 and 10^5, with 50 repetitions for each. The results are shown in Table 3, indicating that increasing the maximum attempts leads to a slight increase in average runtime for both algorithms. This is because: more attempts will increase the computational cost per sun, but also allowing the algorithm has a sufficient opportunity to process challenging yet factorable instances. Since the difference in runtime is very small, we selected 10^5 as the standard value for our experiments.

TABLE 3. Effect of maximum attempts on runtime and success rate (without -1).

Algorithm	Max Attempts	Average Time (s)
Index Calculus	10^4	0.001383
Index Calculus	10^5	0.001559
Random Square	10^4	0.001781
Random Square	10^5	0.001846

3.2. **Experimental Results.** We analyzed the performance of the Index Calculus and Random Square algorithms using datasets generated by SageMath in R. Each algorithm was run on integers of varying bit sizes, and 300 trials were conducted for statistical significance. For Random Squares Algorithm, each integer N was generated as the product of two distinct primes of the given bit size

(see `generate_rsa_keys` in Listing A.1). For Index Calculus Algorithm, the bit size of the prime p was set to twice that of N to ensure a fair comparison, and a generator g of order $p-1$ and a random exponent a were chosen to compute $h = g^a \bmod p$ (see `find_example` in Listing B.1).

For all experiments, we used the following parameter settings based on the preliminary studies: the factor base without -1; the factor base size and number of survivors were set to be 7 and 10, respectively; and the maximum number of attempts was set to 10^5.

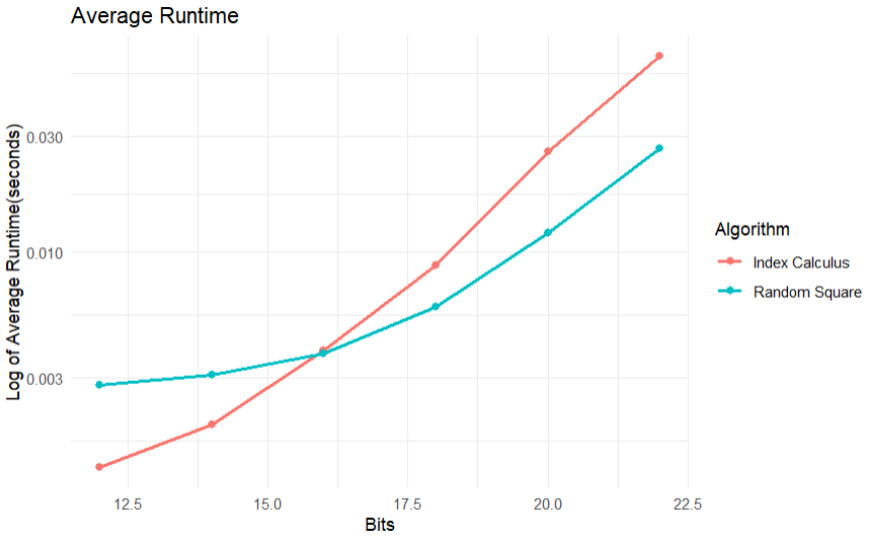

FIGURE 3.1. Average runtime (seconds) vs Bit size for Index Calculus and Random Square algorithms.

Figure 3.1 shows a clear crossing point at a certain bit size, likely around 16 bits. For integers smaller than this size, the Index Calculus algorithm consistently exhibits lower average runtimes, indicating superior efficiency for small scale instances. Conversely, for integers larger than the crossing point, the Random Square algorithm exceeds the Index Calculus method, demonstrating lower average runtime as bit size increases. This crossing point highlights the transition where the relative efficiency of the two algorithms changes.

Figure 3.2 illustrates the distribution of all individual runtimes for each bit size and algorithm. For smaller bit sizes, the runtime variability is relatively high, and the first few boxplots exhibit considerable overlap between the two algorithms. This overlap suggests that although the average runtime of Index Calculus is slightly lower, there is no clear performance advantage in most individual trials. As bit size increases, the difference becomes clearer: the Random Square method not only has lower medians but also consistently narrower distributions, demonstrating more stable and reliable performance compared to the Index Calculus method.

3.3. **Analysis of the results.** In general, the result shows the difference between the Random Square algorithm and the Index Calculus algorithm: a crossover occurs around 16 bit integers. For smaller integers, the Index Calculus algorithm exhibits lower average runtime, while the Random Squares algorithm becomes more efficient for larger integers.

This phenomenon arises from the differing computational characteristics of the two methods. The Index Calculus approach collects a sufficient number of "survivors" that can be completely factored into predefined factor bases, while the Random Square method generates squares randomly and

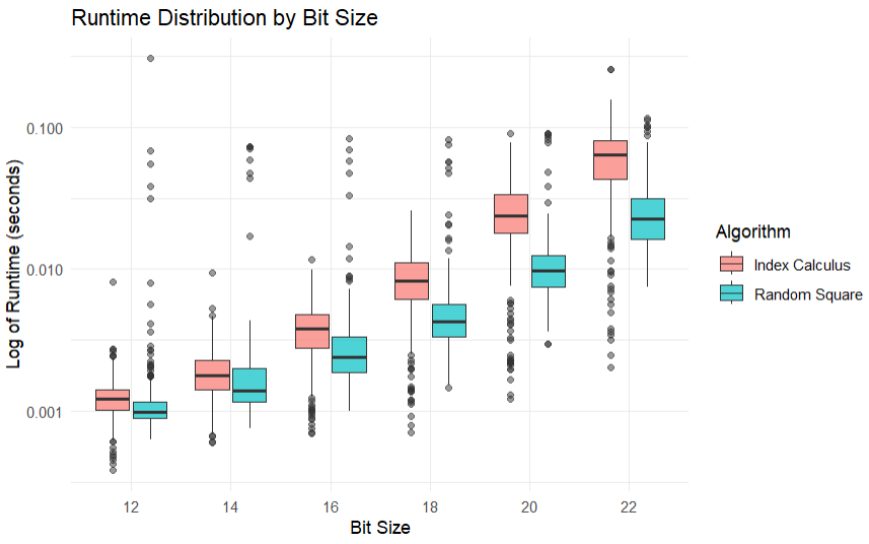

FIGURE 3.2. Runtime distribution by bit size (boxplot) for both algorithms.

attempts to find congruence relationships to obtain "survivors." The former method's computation and verification process is relatively efficient. However, as the bit size increases, the number of candidates requiring testing grows, causing computational costs to rise rapidly. Conversely, while each attempt in the latter method may demand higher computational resources, its probabilistic nature enables it to handle larger integers more efficiently.

A closer look at Figure 3.2 may provide us with more information. For small integers, the Index Calculus algorithm shows lower average runtime, but it also exhibits significant variability between trials. This may stem from the fact that randomly selected exponents at small bit lengths cannot always be fully decomposed by the factor base, leading to longer durations in some trials. On the contrary, the Random Square algorithm shows a more concentrated runtime distribution for small integers. Although the computational cost per trial is slightly higher, its probabilistic characteristics limit the extreme cases. As the integer bit length increases, the Index Calculus must process more candidates, causing its runtime to surge sharply. In the meantime, the Random Square maintains relatively stable performance by finding non-trivial factors through survivors. This explains the crossover point at 16 bit size.

4. CONCLUSION

4.1. **Summary of findings.** Our experiments show a clear crossover around 16-bit integers. For smaller integers, the Index Calculus method has lower average runtimes, while for larger integers, the Random Square method is faster and more stable. This reflects the different computational strategies: the Index Calculus method can efficiently handle small numbers by collecting fully factorizable survivors, while the Random Square method has a higher computational cost, but performs better as the integer size increases.

4.2. **Limitations.** Due to limitations in equipment performance and time constraints, our experiments primarily focused on the efficiency of successful problem instances. Therefore, we recorded the runtime only for 300 instances that were successfully solved by each algorithm. The success

rate is defined as the proportion of randomly generated instances that the algorithm could successfully solve. For the Random Square method, this corresponds to integers N that could be fully factorized, while for the Index Calculus method, it corresponds to DLP problem that could be computed successfully. Not all generated instances were solvable, so only the successful cases were considered in the runtime analysis. Across our experiments, the success rate was relatively low: from 8.93% to 23.81%. This low success rate reflects the difficulty of the respective problems for randomly generated instances.

Therefore, the experiments focus on analyzing the efficiency of successfully solved instances rather than the overall probability of success. This low success probability is primarily caused by two factors. First, larger integers require more "survivors" to establish sufficient relationships, but the small size of factor base limits the number of fully factorizable instances. Second, the fixed number of attempts per candidate restricts the chance of successfully factoring harder instances. Therefore, simple parameter tuning and random attempts cannot guarantee a high success rate for larger integers, and our experiments focus mainly on the efficiency of successful factorizations rather than overall success probability.

5. Acknowledgments

I would like to express my sincere gratitude to Professor Monica Nevins for her guidance and valuable feedback throughout this project. Her insights greatly contributed to the design and analysis of the experiments.

We also gratefully acknowledge the support of modern technology in the preparation of this work. In particular, ChatGPT assisted in generating the tables presented in this paper, while Grammarly was used to correct the grammar of the text.

Appendix A. Full SageMath Code for Random Square Method

Listing 1. generate a proper integer to factor

```
import random
bits_size=6
def generate_rsa_keys(bits=bits_size):
    bound = 2^bits
    lbound = 2^(bits-1)
    p = random_prime(bound, lbound=lbound)
    q = random_prime(bound, lbound=lbound)
    while p == q:
        q = random_prime(bound, lbound=lbound)
    n = p * q
    print(f"n␣=␣p␣*␣q␣=␣{n}")
    return p, q, n
```

Listing 2. Factorization Setup

```
def factor_y(y,fb):
    n=y
    if n==0:
```

```
        return False,None
    if n<0:
        sign=-1
        n=-n
    else:
        sign=1
    v=[0]*len(fb)
    for i,p in enumerate(fb):
        if p==-1:
            if sign==-1:
                v[i]=1
            else:
                v[i]=0
        else:
            count=0
            while n%p==0:
                count+=1
                n=n//p
            v[i]=count
    if n!=1:
        return False,None
    else:
        return True,v
```

Listing 3. Main Random Square Method

```
def main(N):
    print(f"Trying␣to␣factor␣n␣=␣{n}␣=␣{p}*{q}")
    fb=[2,3,5,7,11,13,17]
    m=10
    sur=[]
    print("Generating␣survivors")
    max_attempts = 100000     # attempts to t
    attempts = 0
    while len(sur)<10 and attempts<max_attempts:
        x=random.randint(2,N-1)
        y0=power_mod(x,2,N)
        y=y0
        can_fac,v=factor_y(y,fb)
        if can_fac:
            sur.append((x,y,v))
        attempts += 1
            #print(f"Survivor {len(sur)}: x={x}, y={y}, exp_vec={v}")
    if len(sur) < m:
        return None
    rows=[]
    for(x,y,v)in sur:
        v_mod2=[e%2 for e in v]
        rows.append(v_mod2)
    A=matrix(GF(2),rows)
    #print(A)
    K=A.left_kernel()
```

```
    basis=K.basis()
    if len(basis)==0:
        print("No dependencies found. Need more survivors.")
        return None
    for idx,w in enumerate(basis):
        J=[]
        for i in range(len(sur)):
            if w[i]!=0:
                J.append(i)
        print(f"Indices involved: {J}")
        if not J:
            continue
        X=1
        for i in J:
            x_i=sur[i][0]
            X=(X*x_i)%N
        Z=1
        for i in J:
            y_i=sur[i][1]
            Z *= y_i
        Y=isqrt(Z)
        if Y*Y!=Z:
            #print(f"{Z} is not a perfect square. Skip")
            continue
        #print(f"X = {X}, Y = {Y}")
        g1=gcd(X-Y,N)
        g2=gcd(X+Y,N)
        #print(f"gcd(X-Y, N) = {g1}, gcd(X+Y, N) = {g2}")
        if g1!=1 and g1!=N:
            #print(f"Found factor: {g1}")
            return g1
        if g2 != 1 and g2 != N:
            #print(f"Found factor: {g2}")
            return g2
    #print("Failed to find a factor with the current dependencies.")
    return None
```

LISTING 4. running time check

```
import time
def benchmark_random_square_simple(runs=300):
    times = []
    successful_runs = 0
    while successful_runs < runs:
        p, q, N = generate_rsa_keys()
        start = time.perf_counter()
        result = main(N)
        end = time.perf_counter()
        last=end-start
        if result is not None and result!= 1 and result!= N and N%result==0:
            successful_runs += 1
            times.append(last)
```

```
            print(f"successful runs:{successful_runs}/{runs}, time: {end - start:.6f}")

    avg_time = sum(times)/len(times)
    print(f"Ran {runs} runs. Average time: {avg_time:.6f} seconds")
    return avg_time, times
import pandas as pd
if __name__ == "__main__":
    avg, all_times = benchmark_random_square_simple(300)
df = pd.DataFrame({"Run": range(1, len(all_times)+1), "Time (seconds)": all_times})
display(df)
df.to_excel("random_square_bit12.xlsx", index=False)
```

Appendix B. Full SageMath Code for Index Calculus method

Listing 5. generate an example for DLP

```
bits_size=12
def find_example(bits=bits_size):
    bound = 2^bits
    lbound = 2^(bits-1)
    while True:
        p = random_prime(bound, lbound=lbound)
        N=p-1
        factors=factor(N)
        prime_factors = [f[0] for f in factors]
        max_prime_factor = max(prime_factors)     for g in range(2, p):
                if Mod(g, p).multiplicative_order() == N:
                    a = randint(1, N-1)
                    h = power_mod(g,a,p)
                    return p, g, h, a, N, prime_factors
```

Listing 6. Collect survivors and build linear system

```
def setup_index_calculus(p, g, t, num_survivors=None):
    if num_survivors is None:
        num_survivors = 10
    FB = [2, 3, 5, 7, 11, 13, 17]
    print(f"FB = {FB}")
    survivors = []
    exponent_vectors = []
    a_values = []
    a = 1
    while len(survivors) < num_survivors:
        r = power_mod(g, a, p)
        factors = factor(r)
        if all(prime in FB for prime, exp in factors):
            exp_vector = [0] * t
```

```
            for prime, exp in factors:
                idx = FB.index(prime)
                exp_vector[idx] = exp
            survivors.append(r)
            exponent_vectors.append(exp_vector)
            a_values.append(a)
            print(f"#{len(survivors)}: a={a}, r={r}, vec={exp_vector}")
        a += 1
        if a >= p-1:
            break
    print("Building linear system")
    M = matrix(Zmod(p-1), exponent_vectors)
    v = vector(Zmod(p-1), a_values)
    try:
        dlog_solution = M.solve_right(v)
        dlog_dict = {}
        for i, prime in enumerate(FB):
            dlog_dict[prime] = int(dlog_solution[i])
            print(f"dlog({prime}) = {dlog_dict[prime]}")
        return dlog_dict, FB
    except ValueError as e:
        print(f"Failed: {e}")
        return None, FB
```

Listing 7. Compute discrete logarithm using collected data

```
def compute_discrete_log(h, p, g, dlog_dict, FB, max_attempts=100000):
    t = len(FB)
    for attempt in range(max_attempts):
        a = randint(1, p-2)
        s = (power_mod(g, a, p) * h) % p
        factors = factor(s)
        if all(prime in FB for prime, exp in factors):
            exp_vector = [0] * t
            for prime, exp in factors:
                idx = FB.index(prime)
                exp_vector[idx] = exp
            right_side = 0
            for i, exp in enumerate(exp_vector):
                right_side = (right_side + exp * dlog_dict[FB[i]]) % (p-1)
            dlog_h = (right_side - a) % (p-1)
            if power_mod(g, dlog_h, p) == h:
                print(f"g^{dlog_h} = {h} mod {p}")
            return dlog_h
    print(f"No solution after {max_attempts} attempts")
    return None
```

Listing 8. compute discrete log

```
def complete_index_calculus(p, g, h, t=7, num_survivors=None):
    #print(f"Index Calculus: p={p}, g={g}, h={h}, t={t}")
```

```
    dlog_dict, FB = setup_index_calculus(p, g, t, num_survivors)
    if dlog_dict is None:
        return None
    dlog_h = compute_discrete_log(h, p, g, dlog_dict, FB)
    if dlog_h is not None:
        if power_mod(g, dlog_h, p) == h:
            print("Success")
    else:
        print("Failed")
    return dlog_h
```

Listing 9. running time check

```
import time
def benchmark_index_calculus_random(runs=300):
    times = []
    successful_runs = 0
    while successful_runs < runs:
        p, g, h, true_exp, N, factors = find_example()
        start = time.perf_counter()
        result = complete_index_calculus(p, g, h, t=7, num_survivors=10)
        end = time.perf_counter()
        last = end - start
        if result is not None and result==true_exp:
            successful_runs += 1
            times.append(last)
            #print(f"Run {successful_runs:3d}/{runs}: {last:.6f} seconds")

    avg_time = sum(times) / len(times)
    print(f"Ran␣{runs}␣runs.␣Average␣time:␣{avg_time:.6f}␣seconds")
    return avg_time, times
if __name__ == "__main__":
    avg, all_times = benchmark_index_calculus_random(300)
import pandas as pd
df = pd.DataFrame({"Run": range(1, len(all_times)+1), "Time␣(seconds)": all_times})
display(df)
df.to_excel("factor_base_bit18.xlsx", index=False)
```

References

[1] Monica Nevins, *MAT4360/MAT5160: Mathematical Cryptography*, Preprint, 144pp, 2024.

[2] CoCalc, *SageMath online*, `https://cocalc.com/features/sage`, Last Accessed: November 2025.

[3] Hoffstein, J., Pipher, J., Silverman, J. H., *An Introduction to Mathematical Cryptography*, 2nd ed. Springer, 2014.

[4] Koblitz, N., *A Course in Number Theory and Cryptography*, 2nd ed., Springer-Verlag, Graduate Texts in Mathematics, Vol. 114, 1994.