Comparing AI Algorithms for Optimizing Elliptic Curve Cryptography Parameters in e-Commerce Integrations: A Pre-Quantum Analysis

Felipe Tellez, Jorge Ortíz
Department of Systems and Industrial Engineering, National University of Colombia, Bogotá, Colombia 111321

Abstract—This paper presents a comparative analysis between the Genetic Algorithm (GA) and Particle Swarm Optimization (PSO), two vital artificial intelligence algorithms, focusing on optimizing Elliptic Curve Cryptography (ECC) parameters. These encompass the elliptic curve coefficients, prime number, generator point, group order, and cofactor. The study provides insights into which of the bio-inspired algorithms yields better optimization results for ECC configurations, examining performances under the same fitness function. This function incorporates methods to ensure robust ECC parameters, including assessing for singular or anomalous curves and applying Pollard’s rho attack and Hasse’s theorem for optimization precision. The optimized parameters generated by GA and PSO are tested in a simulated e-commerce environment, contrasting with well-known curves like secp256k1 during the transmission of order messages using Elliptic Curve-Diffie Hellman (ECDH) and Hash-based Message Authentication Code (HMAC). Focusing on traditional computing in the pre-quantum era, this research highlights the efficacy of GA and PSO in ECC optimization, with implications for enhancing cybersecurity in third-party e-commerce integrations. We recommend the immediate consideration of these findings before quantum computing’s widespread adoption.

Keywords—Artificial intelligence; genetic algorithms; particle swarm optimization; elliptic curve cryptography; e-commerce; third-party integrations; pre-quantum computing

I. INTRODUCTION

This paper explores the field of Elliptic Curve Cryptography (ECC), a form of public key cryptography that uses the mathematics of elliptic curves to secure transactions, specifically focusing on its application within e-commerce transactions executed through third-party integrations during a pre-quantum computing era. The aim is to identify the most efficient and effective Artificial Intelligence (AI) algorithm for optimizing parameters essential to ECC’s successful operation within this context. The two leading algorithms being examined are Genetic Algorithms (GA), and Particle Swarm Optimization (PSO).

A. Background

Elliptic Curve Cryptography (ECC), an important and widely used form of public-key cryptography [1][2], provides enhanced security with shorter key lengths [3], making it ideal for resource-constrained environments like e-commerce platforms [34]. Effective ECC operation hinges on the careful selection of parameters such as curve coefficients, base point, prime modulus, and others [4] to [11]. Optimizing these parameters can enhance ECC’s security and efficiency, which is crucial in e-commerce transactions.

Artificial Intelligence (AI) has shown tremendous potential in this parameter optimization [12] to [26]. Notably, two AI algorithms, -GA, and PSO- stand out [12][15]. They represent different subsets and categories of AI algorithms: GA, an Evolutionary Algorithm [18], and PSO, a Swarm Intelligence [19], belong to Population-based Optimization. These algorithms are recognized for their problem-solving and optimization capabilities.

E-commerce transactions often involve integrations with third-party solutions such as Enterprise Resource Planning (ERP) systems, Customer Relationship Management (CRM) systems, payment gateways, data analytics solutions, among others [27][28]. These transactions need to be securely encrypted, making ECC an excellent choice, especially with optimized parameters.

B. Objective

The objective of this paper is to compare the efficacy of GA and PSO in optimizing ECC parameters for e-commerce transactions involving third-party integrations within binary computing. Specifically, the comparison aims to determine which AI algorithm is most efficient in terms of its behavior during the optimization process, and which AI algorithm is most effective in terms of the quality of the results it produces.

C. Scope and Limitations

While ECC can be used in contexts other than e-commerce third-party integrations, such as Web browsers, customer authentication, administrative user authentication, and database persistence, the focus of this article is on transaction integrations with third parties. Interactions between e-commerce systems and backend or third-party solutions such as ERPs, CRMs, payment gateways, and billers are all part of it.

This research uses a simulated e-commerce environment that incorporates a business process that entails creating orders in an emulated ERP. The information for these orders is sent from the e-commerce solution through third-party integrations using web services to imitate real-world conditions. The research relies on API simulations for outbound connections utilizing datasets relevant to these types of scenarios rather than a whole e-commerce solution.

The research also limits its scope to the pre-quantum computing era. Although quantum computing promises significant advances, its implications for ECC and AI algorithms are beyond this study’s scope. The research also excludes other AI
techniques like Simulated Annealing, Ant Colony Optimization or Artificial Neural Networks, despite their applicability to ECC optimization, to keep the research focused.

D. Structure and Contributions

The rest of this paper is organized as follows: Section II reviews related work on ECC optimization, AI techniques, e-commerce integrations, and pre-quantum developments. Section III details materials and methods, including ECC parameters and optimization criteria. Section IV describes the simulation environment design and implementation. Section V presents results and analysis, covering AI algorithm execution, e-commerce simulation, and comparison based on ECC criteria. Section VI discusses future improvements and limitations, including parameter tuning, parallelization, hybrid algorithms, alternative AI techniques, fitness function improvements, diverse cryptographic threats, and quantum computing implications. Section VII concludes with future work and recommendations. Unique contributions include a detailed comparison of GA and PSO for ECC optimization in e-commerce, a novel fitness function, and an evaluation framework for pre-quantum computing.

II. LITERATURE REVIEW

A. ECC Parameter Optimization


A significant aspect of ECC optimization is speeding up point multiplication, a core ECC operation. Hankerson, Menezes, and Vanstone [6] outlined strategies for this focusing on ECC’s computational aspects. Other researchers have examined ECC optimization in MANET and Sensor networks, where hardware plays a significant role. In [7], we can find a state of the art of ECC optimizations in these types of scenarios.

In general, the literature review emphasizes ECC parameter optimization’s importance and complexity, covering domains such as elliptic curves’ mathematical foundations, intricate parameter selection, implementation optimizations, and AI algorithms for multi-objective optimization [8] to [11]. This substantial knowledge forms the foundation of this study.

B. AI Techniques for ECC Parameter Optimization

Artificial Intelligence (AI) exhibits vast potential in ECC parameter optimization, with prominent techniques like Genetic Algorithms (GA) [12] to [14] and Particle Swarm Optimization (PSO) [15] to [17]. These methods contribute distinctive strengths to ECC optimization.

GA, inspired by biological evolution, is known for its effectiveness in exploring complex search spaces, particularly in seeking optimal solutions within intricate landscapes like ECC parameter optimization [18]. On the other hand, PSO, modeled after the social behaviors of birds and fishes, is celebrated for its simple implementation and intrinsic ability to avoid local optima [19]. These methods, by emulating natural processes, present unique solutions to ECC’s challenges, highlighting the connection between nature’s complexity and technological innovation.

Besides these techniques, others like Simulated Annealing (SA) [20], an Stochastic Optimization inspired by the annealing process in metallurgy, is known for adaptability and robustness in solving optimization issues [21], including ECC. Evolutionary Algorithms (EA), similar to GA, involve mechanisms like reproduction and mutation, showing promise in optimizing Elliptic Curves [22]. Machine Learning (ML), where algorithms evolve through data usage, has been applied to Elliptic Curve factorization problems [24]. Also, Tabu Search has been used to enhance ECC operations and multimedia encryption [25] [26].

Each of these named AI methods offers unique benefits in the optimization of ECC parameters; nevertheless, as we mentioned in the introduction section, the focus of our analysis will be on GA, and PSO.

C. E-commerce and Third-party Integrations

The growth of e-commerce has fostered an interconnected technological network involving various third-party entities such as payment gateways, ERP systems, CRM systems, billers, web services, and custom solutions [27] [28]. They exchange vital operational data, including inventory status, billing data, orders information and more.

1) Types of e-commerce integrations: To better understand e-commerce integrations, it’s crucial to consider their directionality, distinguishing between inbound and outbound integrations [29] [30].

- **Inbound Integrations**: SaaS-based e-commerce platforms [31] usually offer an integration layer based on Application Programming Interfaces (APIs). These APIs, typically RESTful web services [32][33], are exposed for third-party consumption. They are provided “out of the box,” ready to be used by external requesters or legacy systems such as ERPs.

- **Outbound Integrations**: These emanate from e-commerce platforms to an external entity and are usually executed through webhooks. These triggers send detailed order information to third-party entities such as the ERP system.

2) The Role of AI in ECC and Third-Party Integrations: As e-commerce evolves, secure and efficient third-party integrations are essential. ECC maintains data security and integrity across these integrations [34]. AI techniques optimize ECC parameters, boosting transaction speed, data security, and overall user experience, offering a competitive edge in e-commerce operations.
D. Pre-Quantum Developments in ECC Optimization

The advent of quantum computing ushers in a new era with its potential to solve complex problems more efficiently than classical computers [35]. However, the implications of this quantum leap for Elliptic Curve Cryptography (ECC) and its parameter optimization using AI algorithms remain largely speculative, as quantum computing is yet to become mainstream. The pre-quantum era, thus, serves as the current framework within which ECC optimization techniques are developed and implemented, focusing on the capabilities of classical computing.

E. Limitations of Similar Research

Although previous studies have made significant contributions to the field of ECC cryptanalysis and security, they primarily focus on specific techniques like Pollard’s Rho, DNA-based cryptography, PSO/Cuckoo Search for key generation, and power optimization for mobile devices. These studies do not explore other AI techniques for optimizing ECC parameters, particularly in the context of e-commerce integrations. Furthermore, there is a lack of consideration for the practical applications of these optimizations in real-world scenarios. The limitations of these studies are summarized in Table I.

<table>
<thead>
<tr>
<th>Study</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>[12]</td>
<td>Focuses on cryptanalysis rather than optimization of ECC parameters for practical applications. Does not explore other AI techniques or their integration with e-commerce systems.</td>
</tr>
<tr>
<td>[13]</td>
<td>Concentrates on multi-cloud security using DNA and HECC techniques but does not explore other AI techniques like GA or PSO for ECC optimization. Lacks practical implementation details for e-commerce integrations.</td>
</tr>
<tr>
<td>[16]</td>
<td>Focuses on mobile devices and optimizing power consumption using PSO and Simplified Swarm Optimization. Does not provide a comprehensive comparison with other AI techniques like GA for ECC optimization. The study’s focus on mobile device constraints limits its applicability to broader e-commerce integrations.</td>
</tr>
<tr>
<td>[17]</td>
<td>Explores PSO and Cuckoo Search Algorithm for ECC key selection but does not provide a comprehensive comparison with other AI techniques like GA. Focuses more on key generation rather than overall ECC parameter optimization in e-commerce contexts.</td>
</tr>
</tbody>
</table>

III. MATERIALS AND METHODS

This section highlights our research methodology, focusing on the ECC parameters to optimize and the criteria for the evaluation of AI techniques.

A. ECC Optimization Parameters

Elliptic Curve Cryptography parameters play distinctive roles, and they can be carefully tuned to improve ECC without sacrificing security. The parameters that will be analyzed for this study are as follows:

1) Choice of elliptic curve: The curve’s equation \( E: y^2 = x^3 + ax + b \) and specific constants \( a \) and \( b \) (curve coefficients) determine the system’s efficiency and security.

2) Field size: Represented by a prime number \( (p) \), the field size affects security and computational load. Larger fields enhance security but need careful balancing with efficiency.

3) Generator point \( G \): The method used for representing points \( (x, y) \) on the curve affects computation speed.

4) Scalar multiplication: Techniques like the Montgomery ladder [36] or sliding window method [37] enhance ECC operations. The operation \( Q = kP \), where \( P \) is a point on the curve and \( k \) is scalar, can be optimized for efficiency.

5) Group order \( n \): This represents the number of points on the elliptic curve and plays a vital role in the security of the ECC system.

6) Cofactor \( h \): The ratio between the number of points on the curve and the group order \( n \). It’s essential in defining the subgroup that is used for cryptographic purposes.

The parameters mentioned above represent only a fraction of the many that can be considered [1] to [11]. Other aspects, such as Hash Function, Pairing Function, Random Number Generation, protocol parameters, use of special curves, batch operations, endomorphism \( (\phi : E \rightarrow E) \), parallelism, efficient arithmetic libraries, hardware acceleration, and more, will not be discussed to maintain the focus of the study.

B. ECC Optimization Criteria

The efficiency and effectiveness, collectively referred to as the efficacy, of the selected AI algorithms in optimizing the ECC parameters, are assessed based on multiple criteria acknowledged as vital evaluation measures by the broader research community. These criteria encompass various aspects that together represent the complete performance of ECC. Below, we outline these criteria:

1) Evaluation of the AI Algorithms (efficiency):
   
   a) Performance: Speed, convergence rate, computational time.
   
   b) Flexibility: Ability to adapt to different problems or changes in the landscape.
   
   c) Robustness: Sensitivity to initial conditions, parameter settings, and noise.
   
   d) Scalability: Ability to handle increasing complexity or problem size.
   
   e) Comparability: Fairness and alignment in comparing the two algorithms.

2) Evaluation of the ECC Parameters Generated (effectiveness):

   a) Security: Resistance against attacks, adherence to cryptographic best practices.
   
   b) Optimality: How close the parameters are to the theoretical best solution.
   
   c) Generalization: Effectiveness across different curve configurations and real-world scenarios.
   
   d) Validity: Compliance with mathematical and cryptographic requirements, such as avoiding singular or anomalous curves.
e) Practicality: Consideration of real-world applications, computational performance, and compatibility with existing systems.

The two aspects of efficiency and effectiveness, are interconnected in efficacy but evaluate different dimensions of the problem. Efficiency focuses on the algorithms themselves and how they perform as optimization techniques [38] to [43], while effectiveness concentrates on the quality and characteristics of the ECC parameters they produce [1] to [11].

IV. SIMULATION ENVIRONMENT DESIGN AND IMPLEMENTATION

The implementation of our simulation consists of an environment of applications and software modules (hereafter referred to as components), built using the Python programming language. These are divided into two main groups: “ECC Params Optimization” and “e-commerce Simulation”. The architecture of this environment is illustrated in Fig. 1.

![Fig. 1. A high-level diagram of the environment’s architecture.](image)

The libraries used in the project include, but are not limited to, ‘numpy’, ‘pandas’, ‘matplotlib’, ‘deap’, ‘gmpy2’, ‘requests’, and ‘tinyec’. These libraries, along with the detailed source code, can be found at: github.com/cftellezc/GA_PSO_ECC_parameter_Optimization.

The components of our simulation environment are described in more detail below.

A. ECC Params Optimization Group

1) Genetic algorithm: The GA.py script or GA App, employs the DEAP (Distributed Evolutionary Algorithms in Python) library to implement a genetic algorithm for ECC parameter optimization. It initiates a population of individuals, where each individual is a list representing potential elliptic curve parameters in ECC. This parameters are the constants $a$ and $b$, the prime number $p$, the generator point $G$ representing points $(x, y)$, the group order $n$ and the cofactor $h$. Through iterative genetic operations like selection, crossover, and mutation, new generations of individuals are produced.

The script uses a custom mutation function that receives the individual, an independent probability $indpb$ (the chance of each attribute to be mutated), and the mutation rate, all of this to mutate the individuals generating prime numbers or perturbing parameters using a Gaussian distribution as follows (Algorithm 1):

**Algorithm 1 Custom Mutation Function**

1: function CUSTOM_MUTATION(individual, indpb, mutation_rate)
2: degree_of_mutation ← 5
3: if mutation_rate > 0.5 then
4: degree_of_mutation ← 10
5: else
6: degree_of_mutation ← 2
7: end if
8: if random value between 0 and 1 < mutation_rate then
9: for $i = 0$ to length of individual − 1 do
10: if random value between 0 and 1 < indpb then
11: if $i = 2$ then
12: individual[$i$] ← generate prime number for $p$
13: else if individual[$i$] is a tuple then
14: individual[$i$] ← find generator point using individual[0], individual[1], individual[2]
15: else
16: individual[$i$] ← individual[$i$] + round(value from Gaussian distribution with mean 0 and standard deviation degree_of_mutation)
17: end if
18: end if
19: end for
20: end if
21: return individual,
22: end function

The script defines several global constants that control the behavior of the genetic algorithm, as summarized in Table II.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP_SIZE</td>
<td>500</td>
<td>Population size</td>
</tr>
<tr>
<td>CXPB</td>
<td>0.5</td>
<td>Crossover probability</td>
</tr>
<tr>
<td>MUTPB</td>
<td>0.2</td>
<td>Mutation probability</td>
</tr>
<tr>
<td>NGEN</td>
<td>40</td>
<td>Number of generations</td>
</tr>
<tr>
<td>MULTIPARENT_CXPB</td>
<td>0.1</td>
<td>Multi-parent crossover probability</td>
</tr>
<tr>
<td>ELITISM_RATE</td>
<td>0.1</td>
<td>Elitism rate</td>
</tr>
</tbody>
</table>

TABLE II. GENETIC ALGORITHM CONSTANTS
They were set up and tuned for better performance using some techniques like grid search. They can be adjusted to tune the performance of the algorithm.

Executed as the main module, the script’s primary function initializes the population, assesses their fitness, and enters a loop for generating and evaluating new individuals over a specified number of generations (NGEN) as follows (Algorithm 2):

Algorithm 2 Main Genetic Algorithm

1: **Initialize**: pop ← toolbox.population()
2: for all individual in pop do
3:   Evaluate fitness and assign to individual
4: end for
5: **Initialize**: elitism_number ← round(len(pop) × ELITISM_RATE)
6: mutation_rate ← MUTPB
7: for g = 0 to NGEN - 1 do
8:   Log: Starting Generation: g + 1
9:   elites ← select top individuals from pop
10: offspring ← select next generation from pop
11: Clone: offspring
12: for i = 0 to len(offspring) - 3 step 3 do
13:   Apply three-point crossover if random() < MULTIPARENT_CXPB
14: end for
15: for all pair child1, child2 in offspring do
16:   Apply crossover to child1, child2 if random() < CXPB
17: end for
18: for all mutant in offspring do
19:   mutate mutant with rate mutation_rate
20: end for
21: Evaluate and set fitness of invalid individuals
22: offspring ← offspring + elites
23: pop ← offspring
24: end for

Summarizing, we initialize the mutation degree utilizing the standard deviation of a Gaussian distribution, and adjusting it according to the mutation_rate. This procedure entails creating random floats to determine the probability of mutation, which is influenced by both the mutation and indpb rates.

Individual parameters are subject to potential mutation, adhering to unique criteria for distinct cases: the third parameter (p) entails deriving a fresh prime number utilizing the BITS_PRIME_SIZE constant — currently set to 256 bits — from the ai_ecc_utils.py module. Tuple parameters representing generator points (G) necessitate fabricating a new point using the prevailing values of a, b, and p.

The remaining parameters (a and b) undergo modifications through Gaussian perturbations, with an imperative to retain integer attributes facilitated by the round function. This mechanism ensures compatibility with DEAP, expecting mutated individuals to be returned as single-item tuples.

The script uses tournament selection and two-point crossover from the DEAP library. It also implements elitism, ensuring that the best individuals from each generation are carried over to the next.

The script evaluates the fitness of the individuals using a function from the ai_ecc_utils.py module (which will be explained later), which calculates the fitness based on the ECC parameters represented by the individual. The script logs the progress of the genetic algorithm, including the statistics of each generation and the best individual from the final generation.

In the context of ECC, the individuals in the population represent different elliptic curves, and the fitness function assesses how well they meet the desired criteria. The genetic algorithm identifies the best-fitting elliptic curve and generates a file named ga_ecc_params.txt, containing the optimal parameters for ECC optimization.

2) Particle swarm optimization: The PSO.py module or PSO App, uses a Particle Swarm Optimization (PSO) algorithm to fine-tune ECC parameters. It initializes particles, where each particle is a list representing potential elliptic curve parameters in ECC, updates their velocities and positions, evaluates fitness, and identifies the best ECC parameter set through the optimal particle. The ECC parameters are the same as those assessed in GA: (a, b, p, G, n, h).

The update_velocity function calculates the new velocity of a particle. It balances global and local exploration using a dynamic inertia weight that linearly decreases from 0.9 to 0.4 over iterations. Two components contribute to the velocity: a cognitive component based on the particle’s best-known position C1, and a social component based on the swarm’s best-known position C2. Special handling is done for the generator point of the elliptic curve, as shown in Algorithm 3.

Algorithm 3 Update Velocity of a Particle

1: **procedure** UPDATEVEL(part, vel, best_part, glob_best_part, iter, max_iter)
2:   new_vel ← []
3:   w_max ← 0.9
4:   w_min ← 0.4
5:   w ← w_max - (w_max - w_min) × iter / max_iter
6: for i = 0 to len(part) - 1 do
7:   v ← vel[i]
8:   r1, r2 ← random(0, 1)
9:   if i = 3 then
10:     cog ← calc_cog(best_part[i], part[i], r1)
11:     soc ← calc_soc(glob_best_part[i], part[i], r2)
12:     if is_tuple(v) then
13:       new_v ← calc_new_v_tuple(v, w, cog, soc)
14:     else
15:       new_v ← calc_new_v(cog, soc)
16:     end if
17:   else
18:     cog ← C1 × r1 × (best_part[i] - part[i])
19:     soc ← C2 × r2 × (glob_best_part[i] - part[i])
20:     new_v ← w × v + cog + soc
21:     if i ≠ 3 then
22:       new_v ← abs(new_v)
23:     end if
24:   end if
25: Append new_v to new_vel
26: end for
27: return new_vel
28: end procedure
The update_position function calculates the new position of a particle based on its velocity. It ensures that coordinates remain positive and integers, and specific attention is given to update the generator point. It also makes use of the early stopping feature implemented in the generator point.

Algorithm 4 Update Position of a Particle

1: procedure UPDATE_POSITION(particle, velocity)
2:     new_particle ← empty list
3:     for i = 0 to len(particle) − 1 do
4:         p ← particle[i]
5:         v ← velocity[i]
6:         if i = 3 then
7:             new_p ← tuple(int(round(p[i] + v[i])) for p[i], v[i]
8:                 in zip(p[:2], v[:2]))
9:         else
10:             new_p ← int(round(p + v))
11:         end if
12:     Append new_p to new_particle
13: end procedure

The script specifies several global constants that regulate how the Particle Swarm Optimization algorithm behaves, as summarized in Table III.

<table>
<thead>
<tr>
<th>Table III. Particle Swarm Optimization Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
</tr>
<tr>
<td>SWARM_SIZE</td>
</tr>
<tr>
<td>MAX_ITERATIONS</td>
</tr>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>MAX_ITERATIONS</td>
</tr>
</tbody>
</table>

The implementation includes a parameter grid for tuning the PSO constants.

The main function initializes the swarm of particles, their velocities, and their best-known positions. Global best-known positions are also identified. The main loop iterates through the swarm, updating velocities and positions using the previously defined functions. Fitness is evaluated for each particle using the same fitness function from the ai_ecc_utils.py module that is used by GA.py (which will be explained later), and best-known positions are updated as necessary. If there is no improvement in global best fitness for 20 iterations, the algorithm stops early, and at the end, statistics regarding fitness values are calculated and printed. The best particle is selected, and its details are printed and written to the pso_ecc_params.txt file.

3) ga_ecc_params.txt: This file contains the best parameters found by the Genetic Algorithm (GA) for ECC parameter optimization.

4) pso_ecc_params.txt: Represents the file with the best ECC parameters found by the Particle Swarm Optimization (PSO) technique.

5) ai_ecc_utils.py: It is a utility module that aids AI algorithms like GA and PSO in the process of ECC parameter optimization. The module’s primary purpose is to facilitate the creation of elliptic curves and their associated parameters, as shown in Algorithm 5.

Algorithm 5 Elliptic Curve Parameter Generation

1: procedure GENERATE_CURVE
2:     signal.signal(signal.SIGALRM, handler)
3:     while True do
4:         p ← get_prime_for_p()
5:     while True do
6:         logging.info("a, b generation")
7:         a ← random.randint(0, p − 1)
8:         b ← random.randint(0, p − 1)
9:         if (4 · a^3 + 27 · b^2) mod p ≠ 0 and
10:             not is_singular(a, b, p) then
11:             break
12:         end if
13:     end while
14:     try:
15:         signal.alarm(TIMEOUT SECONDS)
16:         G ← find_generator_point(a, b, p)
17:         logging.info("G :", G)
18:         signal.alarm(0)
19:         break
20:     except NoGeneratorPointException, TimeoutError:
21:         continue
22:     end while
23:     n ← p − 1
24:     h ← 1
25:     return (a, b, p, G, n, h)
26: end procedure
27: procedure GET_PRIME_FOR_P
28:     return getPrime(BITS PRIME SIZE)
29: end procedure
30: procedure IS_SINGULAR(a, b, p)
31:     discriminant ← (4 · a^3 + 27 · b^2) mod p
32:     return discriminant == 0
33: end procedure
34: procedure FIND_GENERATOR_POINT(a, b, p)
35:     for x in 0 to p − 1 do
36:         rhs ← (x^3 + a · x + b) mod p
37:         if legendre_symbol(rhs, p) == 1 then
38:             y ← tonelli_shanks(rhs, p)
39:             return (x, y)
40:         end if
41:     end for
42: raise NoGeneratorPointException
43: end procedure

It includes functions for generating prime numbers and finding generator points on the elliptic curve using mathematical functions, such as the Legendre symbol and Tonelli-Shanks algorithm [44]. The Legendre symbol determines whether a number is a quadratic residue modulo a prime, essential for finding valid points on the elliptic curve. The Tonelli-Shanks algorithm finds the square root of a number modulo a prime,
crucial for computing the y-coordinates of the points on the curve. These methods ensure the generated points are valid and lie on the elliptic curve, as shown in Algorithm 6.

Algorithm 6 Legendre Symbol and Tonelli-Shanks Algorithm

1: procedure LEGENDRE_SYMBOL(a, p)
2:    ls ← pow(a, (p - 1) / 2, p)
3:    return -1 if ls == p - 1 else ls
4: end procedure
5: procedure TONELLI_SHANKS(n, p)
6:    assert legendre_symbol(n, p) == 1, "n is not a quadratic residue modulo p"
7:    q ← p - 1
8:    s ← 0
9:    while q mod 2 == 0 do
10:       q = q / 2
11:       s = s + 1
12: end while
13: if s == 1 then
14:    return pow(n, (p + 1) / 4, p)
15: end if
16: for z from 2 to p - 1 do
17:    if legendre_symbol(z, p) == -1 then
18:       break
19: end for
20: m ← s
21: c ← pow(z, q, p)
22: t ← pow(n, q, p)
23: r ← pow(n, (q + 1) / 2, p)
24: while t ≠ 1 do
25:    i ← 0
26:    t_i ← t
27:    while t_i ≠ 1 do
28:       t_i = t_i * t
29:       i = i + 1
30: end while
31: b ← pow(c, 2^m - i - 1, p)
32: r ← r * b mod p
33: t ← t * b * b mod p
34: c ← c * b * b mod p
35: m ← i
36: end while
37: return r
38: end procedure

The ai_ecc_utils.py module validates ECC parameters, checking cofactor, prime p, point validity, handling generator point exceptions, matching order and cofactor, and confirming non-singular, anomalous, or supersingular characteristics [45], as shown in Algorithm 7. These validations ensure the integrity and security of the ECC parameters used in the cryptographic system.

The module also implements Pollard’s rho attack [45] to evaluate the security of the generated ECC parameters. This attack is a well-known method for finding discrete logarithms in elliptic curves, making it an essential tool for assessing the resilience of the cryptographic system against specific types of attacks. It employs functions to add two points on an elliptic curve, apply the “double and add” method for point multiplication, and check if a point is “distinguished” point multiplication, and check if a point is “distinguished.

Algorithm 7 Validation and Properties of Elliptic Curve

1: procedure VALIDATE_CURVE(a, b, p, G, n, h)
2:    if h < 1 then
3:        log "The cofactor h is less than 1, which makes it invalid.”
4:        return False
5:    end if
6:    if p == 0 then
7:        log "The prime p can’t be zero.”
8:        return False
9:    end if
10: if len(G) == 2 then
11:    x, y ← G
12:    if (y - x * x - a * x - b) mod p ≠ 0 then
13:        log "The point G is not on the curve!”
14:        return False
15:    end if
16:    field ← SubGroup with (p, G, n, h)
17:    if No generator point in field then
18:        log "No generator point found!”
19:        return False
20:    end if
21:    curve ← Curve with (a, b, field, "random_curve")
22: else
23:    log "Invalid generator point provided. Skipping curve creation.”
24:    return False
25: end if
26: order ← n
27: if h ≠ field.h then
28:    log "The cofactor does not match the expected cofactor!”
29:    return False
30: end if
31: if IS_SINGULAR(a, b, p) then
32:    log "The curve is singular!”
33:    return False
34: end if
35: if IS_ANOMALOUS(p, order) then
36:    log "The curve is anomalous!”
37:    return False
38: end if
39: if IS_SUPERSINGULAR(p, order) then
40:    log "The curve is supersingular!”
41:    return False
42: end if
43: return True
44: end procedure
45: procedure IS_SINGULAR(a, b, p)
46:    discriminant ← (4 * a^3 + 27 * b^2) mod p
47:    return discriminant == 0
48: end procedure
49: procedure IS_ANOMALOUS(p, n)
50:    return p == n
51: end procedure
52: procedure IS_SUPERSINGULAR(p, n)
53:    if p ∈ [2, 3] or not isprime(p) then
54:        return False
55:    end if
56:    return (p + 1 - n) mod p == 0
57: end procedure
Lastly, the ai_ecc_utils.py module calculates the fitness function, incorporating all the elliptic curve validations. It evaluates the fitness of a candidate, whether an “individual” in the GA population or a “particle” in the PSO swarm, as shown in Algorithm 8.

Algorithm 8 Pollard’s Rho Attack on an Elliptic Curve

1: function P_RHO_ATTACK(G, a, b, p, order, t, max_iter)
2: Q_a, Q_b ← G, G
3: a, b ← 0, 0
4: power_of_two ← 1
5: iterations ← 0
6: while iterations < max_iterations do
7:     for i in range(power_of_two) do
8:         i ← Q_a[0] mod 3
9:         if i = 0 then
10:            Q_a ← add_points(Q_a, G, a, p)
11:            a ← (a + 1) mod order
12:         else if i = 1 then
13:            Q_a ← double_and_add(2, Q_a, a, p)
14:            a ← (2 · a) mod order
15:         else
16:             Q_a ← double_and_add(2, Q_a, a, p)
17:             a ← (2 · a) mod order
18:            Q_a ← add_points(Q_a, G, a, p)
19:             a ← (a + 1) mod order
20:         end if
21:         if is_distinguished(Q_a, t) then
22:             return a, Q_a
23:         end if
24:     end for
25:     for i in range(2) do
26:         Repeat the same steps for Q_b, but twice per iteration
27:     end for
28:     iterations ← iterations + 1
29:     if Q_a = Q_b then
30:         power_of_two ← power_of_two × 2
31:         Q_b ← Q_a
32:         b ← a
33:     end if
34: end while
35: logging.info("No collision found within the")
36: specified maximum number of iterations.")
37: return None
38: end function

The fitness function extracts the elliptic curve parameters (a, b, p, G, n, h). Curve Validation checks if the parameters form a valid curve, returning a fitness of 0 if not. The expected order of the curve is calculated, checks Hasse’s theorem bounds [44] and the Hasse score is computed to evaluate how close the actual order is to the expected. Pollard’s Rho Attack is attempted, with a longer execution time indicating higher resistance. An Attack Resistance Score is assigned. The Final Fitness Calculation includes 40% weight to the natural logarithm of the curve’s order, 20% to the Hasse score (weighted by the logarithm of the order), 20% to the execution time of the attack, and 20% to the resistance score.

Algorithm 9 Function to evaluate the fitness of a candidate in GA or PSO

1: function EVALUATE(candidate)
2:     Extract a, b, p, G, n, h from candidate
3:     if not VALIDATE_CURVE(a, b, p, G, n, h) then
4:         return 0
5:     end if
6:     expected_order ← p + 1 – 2 · √p
7:     upper_bound ← expected_order + 2 · √p
8:     hasse_score ← max(0, upper_bound – lower_bound)
9:     order ← current time
10:    rho_attack_result ← P_RHO_ATTACK(G, a, b, p, expected_order)
11:    execution_time ← current time – start_time
12:    max_time ← 10.0, min_time ← 0.1
13:    execution_score ← max(0, min(1, (max_time−min_time)/max_time))
14:    attack_resistance_score ← 1 if rho_attack_result is None
15:    fitness ← 0.4 · log(n) + 0.2 · hasse_score · log(n) + 0.2 · execution_score + 0.2 · attack_resistance_score
16:    return fitness
17: end function

These utility functions facilitate key generation by creating a random private key, an integer within the range [1, n-1], and producing the corresponding public key, which is computed by cumulative fitness score is returned, reflecting the candidate’s elliptic curve suitability.

These are several global constants that are defined for ai_ecc_utils.py module, as summarized in Table IV:

TABLE IV. AI_ECC_UTILS.PY MODULE CONSTANTS

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITS_PRIME_SIZE</td>
<td>256</td>
<td>Size of the prime in bits. Generates a n-bit prime number</td>
</tr>
<tr>
<td>POLLARDS_RHO_TRIALS</td>
<td>20</td>
<td>Number of trials for the Pollard’s Rho function</td>
</tr>
<tr>
<td>POLLARDS_RHO_MAX_ITER</td>
<td>10**2</td>
<td>Maximum iterations for each trial in the Pollard’s Rho function</td>
</tr>
</tbody>
</table>

This module is designed to be used in conjunction with other modules that implement bio-inspired algorithms, such as PSO and GA, and given how it was designed, other AI algorithms that are capable of optimizing elliptical curves may use it in the future.

B. E-commerce Simulation Group

1) well-known_curves_params.txt: Represents parameters for standard elliptic curves used in cryptography, such as secp256k1 and brainpoolP256r1 [46].

2) ecc.py: This utility module includes classes and functions for elliptic curve cryptography. A key function reads ECC parameters from files like ga_ecc_params.txt, pso_ecc_params.txt, secp256k1.txt, or brainpoolP256r1.txt (the latter two as well-known_curves_params.txt), creating a structured set of parameters to be used in the e-commerce simulation, as shown in Algorithm 10.

The ecc.py module has support functions like ec_addition for adding points on an elliptic curve, and ec_scalar_multiplication for multiplying a point by a scalar using the double-and-add method, as shown in Algorithm 11.

These utility functions facilitate key generation by creating a random private key, an integer within the range [1, n-1], and producing the corresponding public key, which is computed by
Algorithm 10 Initialization of ECC Parameters

1: function INITIALIZE_PARAMS(option)
2: Define filename based on option:
3: if option == "1" then
4: filename ← "ga erectile_params.txt"
5: else if option == "2" then
6: filename ← "pso erectile_params.txt"
7: else if option == "3" then
8: filename ← "secp256k1.txt"
9: else if option == "4" then
10: filename ← "brainpoolP256r1.txt" end if
11: Open filename for reading as f
12: Initialize params_dict as an empty dictionary
13: for each line in f do
14: Split line into key, value and store in params_dict
15: Convert value to integer
16: end for
17: Extract parameters p, a, b, G, n, h from params_dict
18: G ← ECPoint(Gx, Gy)
19: return ECCParameters(p, a, b, G, n, h)
20: end function

Algorithm 11 Point Addition and Scalar Multiplication

1: function EC_ADDITION(P, Q, p)
2: if P is None or inf then return Q
3: end if
4: if Q is None or inf then return P
5: end if
6: if P.x == Q.x then
7: if P.y = -Q.y mod p then return ECPoint(None, None)
8: else
9: m ← (3P.x^2 + a) / (2P.y)^1 mod p
10: end if
11: m ← (Q.y - P.y) / (Q.x - P.x)^1 mod p
12: end if
13: x ← m^2 - P.x - Q.x mod p
14: y ← m(P.x - x) - P.y mod p return ECPoint(x, y)
15: end function

Algorithm 12 Private and Public Key Generation

1: function GENERATE_PRIVATE_KEY(params)
2: return randbelow(params.n - 1)
3: end function
4: function GENERATE_PUBLIC_KEY(private_key, params)
5: result ← ec_scalar_multiplication(params.G, private_key)
6: private_key, params)
7: return result
8: end function

Additionally, within the ecc.py module, the functions ec_addition and ec_scalar_multiplication are utilized to encrypt and decrypt messages, as shown in Algorithm 13.

Lastly, ecc.py uses the hmac Python library to generate and verify a Hash-based Message Authentication Code (HMAC) for a given message and key, ensuring the integrity and authenticity of messages.

3) Orders dataset: This e-commerce dataset includes invoices generated by an authorized online retailer [47]. These have been pre-processed and curated as orders for practical simulation purposes, converting them into order data. This data feeds Entity A and is then sent to Entity B in the e-commerce simulation.

4) EntityA.py: This component emulates an e-commerce solution, and communicates with a simulated ERP server (EntityB.py) via Elliptic Curve Cryptography (ECC). It includes functionalities for server connections, ECC parameters, key management, order message generation, and transaction handling. The system reads data from a spreadsheet file (Orders dataset), encrypts messages with ECC and HMAC using the module ecc.py, sends them to the ERP server, and employs Elliptic Curve Diffie-Hellman (ECDH) for key agreement, as shown in Algorithm 14.

5) EntityB.py: It is a simulated ERP server that interacts with the emulated e-commerce solution (EntityA.py) using Flask (Python web framework). It handles orders, key retrievals, initiates ECC and ECDH keys, and allows users to select the type of ECC parameters (such as GA, PSO, or well-known curves) to be used throughout the simulation. Functions for decrypting orders and verifying MACs are included.

The ECC parameters employed in the simulation are selected by the user and loaded from the corresponding txt file. The ecc.py module handles all cryptographic operations throughout the process.

6) pollards_rho_attack.py: To evaluate the ECC parameters in the simulation, a component is designed to attack the communication between EntityA.py and EntityB.py. This Python script executes Pollard's rho attack [36][45] on the e-commerce simulation, employing the "tortoise and hare" technique to implement the attack logic. By leveraging multi-processing for parallelization and handling collisions to determine the private key, it interacts with the ERP server (EntityB.py) to gather essential data such as the public key of the targeted entity. The script employs various methods like multiplying the base point G by the private key, as shown in Algorithm 12.
Algorithm 13 ECC Encryption and Decryption
1: function ENCRYPT_MESSAGE(message, public_key, params)
2: if not is_valid_point(public_key, params) then
3: raise ValueError("Public key is not a valid point on the elliptic curve")
4: end if
5: k ← generate_private_key(params)
6: if not is_valid_scalar(k, params) then
7: raise ValueError("Invalid scalar value")
8: end if
9: C1 ← ec_scalar_multiplication(params.G, k, params)
10: C2 ← ec_scalar_multiplication(public_key, k, params)
11: message_bytes ← message.encode(utf-8)
12: encrypted_message ← []
13: for byte in message_bytes do
14: if C2.x is None then
15: raise ValueError("Encryption failed: kQ resulted in the point at infinity")
16: end if
17: encrypted_byte ← byte ⊕ (C2.x&0xFF)
18: encrypted_message.append(encrypted_byte)
19: end for
20: return C1, encrypted_message
21: end function

Algorithm 14 Main Function Procedure for Entity A
1: procedure MAIN
2: Initialize ServerConnection with server URL
3: Initialize ECCParams with ServerConnection
4: Request ECC parameters from server
5: Initialize ECCKeys with ECC parameters
6: Generate ECC private and public keys
7: Initialize RetailMessage with MS Excel file path
8: Initialize TransactionManager with necessary objects
9: Run transactions until a predetermined end time
10: end procedure
Algorithm 15 Pollard’s Rho Attack on Elliptic Curve Cryptography

1: procedure P_RHO(init_value, G, public_key, params, manager_dict)
2: \hspace{10pt} Print start message with init_value
3: \hspace{10pt} tortoise ← ec_scalar_multiplication(G, init_value, params)
4: \hspace{10pt} hare ← tortoise
5: \hspace{10pt} tortoise_scalar ← hare_scalar ← init_value
6: \hspace{10pt} for i = 1 to 2 x params.n + 1 do
7: \hspace{20pt} tortoise, tortoise_scalar ← step(tortoise, tortoise_scalar, G, public_key, params)
8: \hspace{20pt} hare, hare_scalar ← step(hare, hare_scalar, G, public_key, params)
9: \hspace{20pt} hare, hare_scalar ← step(hare, hare_scalar, G, public_key, params)
10: \hspace{20pt} if tortoise = hare and tortoise \neq None then
11: \hspace{30pt} scalar_difference ← gmpy2.f_mod((tortoise_scalar - hare_scalar), params.n)
12: \hspace{30pt} if scalar_difference = 0 then
13: \hspace{40pt} continue
14: \hspace{30pt} scalar_difference_inverse ← gmpy2.invert(scalar_difference, params.n)
15: \hspace{30pt} secret_key ← gmpy2.f_mod((scalar_difference_inverse * hare_scalar), params.n)
16: \hspace{20pt} if manager_dict[‘found_flag’] then
17: \hspace{30pt} manager_dict[‘found_flag’] ← True
18: \hspace{30pt} Print found secret key
19: \hspace{20pt} break
20: \hspace{20pt} end if
21: \hspace{20pt} end if
22: \hspace{20pt} end for
23: \hspace{10pt} Print result message
24: \hspace{10pt} end procedure
25: procedure S_TEP(point, scalar, G, public_key, params)
26: \hspace{10pt} Define a function to move and update point and scalar
27: \hspace{10pt} Handle different cases based on x-coordinate of point
28: \hspace{10pt} return new point and corresponding scalar
29: \hspace{10pt} end procedure
30: function GET_ECC_PARAMS_FROM_SERVER
31: \hspace{10pt} Retrieve ECC parameters from server
32: \hspace{10pt} end function
33: function GET_PUBLIC_KEY_FROM_ENTITY_B
34: \hspace{10pt} Retrieve public key from Entity B
35: \hspace{10pt} end function
36: procedure POLLARDS_RHO_ATTACK_ON_ENTITY_B(params)
37: \hspace{10pt} Carry out Pollard’s rho attack on Entity B in parallel
38: \hspace{10pt} end procedure
39: procedure MAIN
40: \hspace{10pt} Retrieve ECC parameters
41: \hspace{10pt} Initialize parameters
42: \hspace{10pt} Carry out Pollard’s rho attack on Entity B
43: \hspace{10pt} end procedure
44: if name = ”main” then
45: \hspace{10pt} Call Main()
46: \hspace{10pt} end if

convergence. In these graphs, the average fitness value for PSO seems to degrade over time, despite optimizing the Cognitive and Social parameters using grid search. Conversely, GA consistently refines its fitness function, demonstrating a more stable trend.
The subsequent charts illustrate these trends in Fig. 3 and Fig. 4:

![Fitness Progression over Generations - GA](image)

**Fig. 3. Fitness progression over generations - GA.**

It’s worth noting that both algorithms are utilizing the same fitness function. On the other hand, GA generates results much faster than PSO, even though PSO has a feature that allows for early stopping if there’s no improvement in the global best fitness for 20 iterations. While GA completes its optimization task in roughly 2 minutes, PSO takes about 22 minutes to finish its task. The tests were conducted on a 2019 15-inch MacBook Pro with the following specifications: Processor: 2.3 GHz 8-Core Intel Core i9; RAM: 16 GB 2400 MHz DDR4; Graphics: Radeon Pro 560X 4 GB and Intel UHD Graphics 630 1536 MB.

**B. Execution of e-commerce Integration Simulation**

In the simulation testing the transmission of order information in third-party integrations, from the simulated e-commerce to the ERP, the results are promising for both GA and PSO when compared to well-known curves like secp256k1.txt and brainpoolP256r1.txt. The speed of encrypting and decrypting messages was nearly identical among the four curve types, ranging from fractions of a second to less than 2 seconds. Moreover, a successful Pollard’s Rho attack could not be executed in any scenario, given the computationally intensive task of determining the private key, even when the simulation ran for a week. This approach emphasizes frequent changes in ECC parameters. Compared to well-known curves, GA and PSO offer an advantage by utilizing novel curves without known values for attackers.

**C. Comparison based on ECC Optimization Criteria**

The table below displays the winning algorithm based on the acceptance criteria previously outlined in this paper, as shown in Table VII:

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Winning Algorithm</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficiency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td>GA</td>
<td>Faster generation of ECC parameters.</td>
</tr>
<tr>
<td>Flexibility</td>
<td>GA</td>
<td>Consistent evolution of the fitness function even when increasing the number of generations.</td>
</tr>
<tr>
<td>Robustness</td>
<td>GA</td>
<td>Greater tolerance to noise in initial constants definition.</td>
</tr>
<tr>
<td>Scalability</td>
<td>GA</td>
<td>Remains faster and more efficient even when increasing generations and parallelizing calculations.</td>
</tr>
<tr>
<td>Comparability</td>
<td>GA</td>
<td>Lower computational complexity.</td>
</tr>
<tr>
<td>Effectiveness</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td>Both GA and PSO</td>
<td>Neither faced successful attacks in the e-commerce integration scenario. However, during parameter generation, PSO experienced more successful attacks than GA when searching for point collisions.</td>
</tr>
<tr>
<td>Optimality</td>
<td>GA</td>
<td>Consistently closer to expected maximum values and showed improvement over time.</td>
</tr>
<tr>
<td>Generalization</td>
<td>Both GA and PSO</td>
<td>Quick in both encryption and decryption processes in the simulated scenario, and neither faced successful attacks.</td>
</tr>
<tr>
<td>Validity</td>
<td>Both GA and PSO</td>
<td>ECC parameters were free from singular or anomalous curves.</td>
</tr>
<tr>
<td>Practicality</td>
<td>GA</td>
<td>Superior in performance while being compatible with real-world existing systems.</td>
</tr>
</tbody>
</table>

![Fitness Progression over Iterations - PSO](image)

**Fig. 4. Fitness progression over iterations - PSO.**
Based on the previous table and the information in this section, Genetic Algorithms (GA) outperform Particle Swarm Optimization (PSO) in ECC parameter optimization, not just for third-party e-commerce integrations but also, in terms of efficiency, across any Elliptic Curve Cryptography scenario.

VI. FUTURE IMPROVEMENTS AND KNOWN LIMITATIONS

Here are some areas we believe can be improved in the future. These improvements can be considered as potential next steps or future work:

A. Advanced Parameter Tuning

The current scripts employs grid search for fine-tuning the initial constants. Employing intricate parameter tuning techniques like random search, Bayesian optimization, or metaheuristic algorithms might pinpoint superior parameter values, enhancing the efficacy of the GA and PSO algorithms.

B. Parallelization

For PSO, the fitness evaluation along with position and velocity updates for individual particles can be executed concurrently, given their independence. Incorporating parallel processing could drastically curtail computational duration, particularly with larger swarms or increased iteration counts.

C. Hybrid Algorithms

Merging PSO with alternative optimization algorithms can spawn a hybrid model that capitalizes on the strengths of each algorithm. For instance, integrating GA to evolve the swarm while utilizing PSO for refining solutions could augment solution quality and the robustness of the optimization.

D. Exploration of Alternative AI Techniques

In light of the promising results yielded by the GA and PSO algorithms in this research, it stands to reason that the exploration of alternative artificial intelligence techniques could further optimize ECC parameter generation. Leveraging the same fitness function and utility components established in this study would serve as a foundational bedrock for the integration of techniques such as deep learning, reinforcement learning, or swarm intelligence variations different from PSO, facilitating a seamless transition and a consistent basis for performance evaluation. Such endeavors could potentially unearth novel approaches that are more efficient, secure, and robust, pushing the boundaries of what can be achieved in cryptographic parameter optimization and ensuring a forward momentum in ECC security research.

E. Improved Fitness Function

The ai_ecc_utils.evaluate function, employed as the script’s fitness function, ascertains the security of an elliptic curve. This function could undergo enhancement or be supplanted with an alternate fitness function to steer the PSO algorithm search more effectively. For instance, the fitness function might integrate additional security parameters or be tailored to prioritize specific elliptic curve types.

F. Integrating Diverse Cryptographic Threat Evaluations

In the future, addressing diverse threats to elliptic curves beyond just Pollard’s rho attack is crucial. With historical cryptographic vulnerabilities exposed by techniques like the Pohlig-Hellman method and Baby-step Giant-step [45], and the impending rise of quantum computing introducing threats like Shor’s algorithm, modern ECC methods may be at risk [35]. Integrating and evaluating these varied attacks in the fitness function will be essential for fortified defenses.

G. Quantum Computing Implications

It’s crucial to consider the advent of quantum computers and their potential impact on cryptographic algorithms [35]. As quantum computing technology evolves, both GA and PSO algorithms’ performance and security measures need re-evaluation to ensure they remain resilient against quantum threats. Additionally, the designed fitness function for this study, which evaluates the security and efficiency of ECC parameters, could be implemented in quantum environments since it is based on optimization through artificial intelligence, allowing for the study of its behavior in such contexts.

VII. CONCLUSIONS

In light of our comprehensive research and systematic evaluation, it is clear that Genetic Algorithms (GA) are more efficient than Particle Swarm Optimization (PSO) in the optimization of Elliptic Curve Cryptography (ECC) parameters. This assertion is based on the adept fitness function we designed and utilized in our research. GA’s superiority is evident in third-party e-commerce integrations. Both algorithms are robust, successfully withstanding attacks in e-commerce integration tests. Nevertheless, GA consistently delivers quicker and more reliable performance, integrating seamlessly with real-world systems. This benefit, along with its efficient evolution and rapid ECC parameter generation, underscores GA’s dominance in this field. While PSO does offer distinct advantages, its potential is hampered by its relative inefficiency, especially regarding computational speed. Consequently, we strongly advise stakeholders aiming to optimize ECC parameters to prioritize the GA approach.

REFERENCES


