A novel cuckoo search strategy for automated cryptanalysis: a case study on the reduced complex knapsack cryptosystem

Article *in* International Journal of Systems Assurance Engineering and Management · November 2017

DOI: 10.1007/s13198-017-0690-9

CITATIONS
11

READS
120

2 authors:

Ashish Jain
Manipal University Jaipur
20 PUBLICATIONS **149** CITATIONS

SEE PROFILE

Narendra Chaudhari
Visvesvaraya National Institute of Technology
231 PUBLICATIONS **1,404** CITATIONS

SEE PROFILE

Int J Syst Assur Eng Manag
https://doi.org/10.1007/s13198-017-0690-9

ORIGINAL ARTICLE

A novel cuckoo search strategy for automated cryptanalysis: a case study on the reduced complex knapsack cryptosystem

Ashish Jain[1] · **Narendra S. Chaudhari**[1,2]

Received: 6 March 2017 / Revised: 2 August 2017

Abstract During the past decade several new variants of knapsack cryptosystems have been reported in the literature. Hence, there is a growing demand for automated cryptanalysis of knapsack cryptosystems. Brute force approach is capable to cryptanalyze simple stages of cryptosystems while cryptanalysis of complex cryptosystems demands efficient methods and high-speed computing systems. In the literature, several search heuristics have proven to be promising and effective in automated cryptanalysis (or attacks) of classical or reduced cryptosystems. This paper presents the automated cryptanalysis of the reduced multiplicative knapsack cryptosystem using three different search heuristics, namely, cuckoo search, particle swarm optimization and genetic algorithm. It should be noted that the considered cryptosystem is reduced but is complex and practical representative. To the best of our knowledge, this is the first time when the cuckoo search is utilized for automated cryptanalysis of the complex cryptosystem. The performance of developed techniques has been measured in terms of time taken by the algorithm (i.e., how efficient the algorithm is?), number of times the original plaintext is determined (i.e., success rate), and the number of candidate plaintexts is examined before determining the original plaintext (i.e., how effective the algorithm is?). For the case considered, performance of the proposed techniques, namely, novel binary cuckoo search (NBCS), improved genotype–phenotype binary particle swarm optimization (IGPBPSO), and new genetic algorithm (NGA) is as follows: roughly the NBCS technique is 12% and 8% more efficient, 6% and 5% more successful, and 16% and 12% more effective than IGPBPSO and NGA, respectively. This results show that the proposed NBCS strategy is superior to IGPBPSO and NGA, and therefore NBCS strategy can be used as an efficient and effective choice for solving similar binary discrete problems such as 0–1 knapsack problem, set covering problem, etc.

Keywords Swarm intelligence · Cuckoo search · Evolutionary computing · Genetic algorithm · Automated cryptanalysis · Knapsack cryptosystems

1 Introduction

Security of information is a major aspect in most of the applications that can be achieved by three fundamental core components: confidentiality, integrity and availability. For achieving confidentiality over the insecure communication channel, cryptosystems are employed that perform encryption and decryption with the use of a secret key (Stinson 2005; Menezes et al. 2010; Martin 2017). There are two divisions of cryptosystems: symmetric key cryptosystems and asymmetric (or public) key cryptosystems. Knapsack cryptosystems belong to the class of public key cryptosystems in which a public key is used for encryption and a private key is used for decryption. The process of finding flaws or oversights in the design of cryptosystem is

✉ Ashish Jain
phd11120101@iiti.ac.in; ashishjn.research@gmail.com

Narendra S. Chaudhari
nsc@iiti.ac.in; nsc0183@yahoo.com

[1] Discipline of Computer Science and Engineering, Indian Institute of Technology (IIT) Indore, Indore, India

[2] Discipline of Computer Science and Engineering, Visvesvaraya National Institute of Technology (VNIT) Nagpur, Nagpur, India

Published online: 21 November 2017

called cryptanalysis. The goal of the adversary (or cryptanalyst) is to systematically recover original text (plaintext) from unreadable text (ciphertext) by mounting an attack on the cryptosystem. There are several modes of attacks depending on the information available to the attacker: known-plaintext attack, chosen-plaintext attack, adaptive chosen-plaintext attack, chosen-ciphertext attack, adaptive chosen-ciphertext attack and ciphertext-only attack (Menezes et al. 2010; Martin 2017). A ciphertext only attack is one where the objective of the cryptanalyst is to recover the plaintext or deduce decryption key by observing only the known ciphertexts. Any cryptosystem vulnerable to this type of attack is believed to be completely insecure. This type of attack is most challenging (Menezes et al. 2010; Martin 2017), and we demonstrate such attack on the reduced complex knapsack cryptosystem. Merkle and Hellman (1978) have proposed the first public key cryptosystem (PKC). This particular PKC is based on the knapsack problem (an NP-complete problem). Due to this reason, the cryptosystem which is based on the concept of Merkle and Hellman is referred to as knapsack cryptosystem. On the other side, many existing PKCs (e.g., RSA) are based on the intractable integer factorization and discrete logarithm problems. However, Shor (1997) has proposed quantum algorithm for solving integer factorization and discrete logarithm problems. Hence, there is a growing demand of post-quantum PKCs because many existing PKCs based entirely on integer factorization and discrete logarithm problems are breakable using Shor's algorithm on a quantum computer. Knapsack cryptosystems are based on the NP-complete problem and are much faster than the RSA scheme. Therefore, it might be a viable option, especially for low constraint devices such as cellular devices and radio frequency identification tags (Kate and Goldberg 2011). In this regard, during the past decade several new variants of the knapsack cryptosystem have been proposed in the literature, for example, Wang et al. 2007; Wang and Hu 2010; Kate and Goldberg 2011; Hei and Song 2014. Hence, there is a growing demand for automated cryptanalysis of the knapsack cryptosystems. It should be noted that cryptosystems are developed in such a way that an exhaustive computer search of the secret key in their key space (or solution space) becomes impractical (Stinson 2005; Martin 2017). This implies that cryptanalysis problems are combinatorics search problems where the search space is all possible combinations of key elements. The cryptanalytic attack via exhaustive (brute-force) searching can be used against any encrypted text, however, in the worst case this will traverse the entire search space (Stinson 2005; Martin 2017). For instance, if the key size of a knapsack cryptosystem is 64-bit length, then there are 2^{64} distinct possible keys. If a supercomputer is available that could verify 1 billion keys per second will require about 584 years. Hence, these ciphers remain secure from such an attack, since the key space size is such that the resources and time are not available for searching the key exhaustively. On the other hand, computational intelligence techniques (e.g., search heuristics or metaheuristics) are capable to reduce the search space efficiently to a considerable extent (Laskari et al. 2007; Danziger and Henriques 2012; Awad and El-Alfy 2015). This paper utilizes three different search heuristics for automated cryptanalysis of the reduced complex knapsack cryptosystem.

Basics about proposed work: Assuming that we have a ciphertext of n bits length, the corresponding plaintext will also be of n bits length. If the exhaustive search is carried out, then in the worst case 2^n candidate plaintexts are needed to be examined. Instead, if search heuristic is applied, the exact plaintext can be determined by exploring only 10–20% plaintexts in the search space. This is possible because an important component of the search technique is fitness function which takes a candidate plaintext as input and notify how "good" our solution is with respect to the problem. Candidate plaintext means a possible plaintext which is evolved by the search technique by searching the plaintext space. The fitness values are used in a process of natural selection to select which potential plaintexts will continue in the next generation, and which will die. It should be noted that the natural selection process does not merely choose the top x number of solutions; the solutions are instead chosen statistically in a way, it is more likely that a solution with a higher fitness value will be chosen. For creating the necessary diversity in the population, variation operators (e.g., crossover and mutation) are used that create new solutions from existing ones. Fine tuning of variation operators is also an important step for recovery of the plaintext in less iterations (Danziger and Henriques 2012).

1.1 Literature review

Computational intelligence (CI) is a well-established set of computational techniques or search heuristics with new theories and sound nature-inspired concepts. The search techniques use characteristics of intelligent systems to effectively solve search and optimization problems. Recently, fitness based particle swarm optimization (Sharma et al. 2015), and artificial bee colony algorithm with global and local neighborhood (Jadon et al. 2014) strategies have been reported in the literature to improve the charateristics of the swarm intelligence based algorithms. The last decade has witnessed evolutionary algorithms, swarm intelligence and hybrid techniques are successfully utilized for optimization of various engineering problems, for example, a new hybrid immune and hill climbing local search technique has been developed by

Yildiz (2009a) and Yildiz (2009b) for optimizing design and manufacturing machining parameters. Hill climbing algorithm has also been hybridized with simulated annealing to optimize design and manufacturing problems (Yildiz 2009c). Structural design problem has been significantly optimized using hybrid differential evaluation and taguchi method (Yildiz 2013a), and using hybrid interior search-hill climbing algorithm (Yildiz 2017). A significant study on structural optimization of vehicle components has been carried out by Yildiz et al. (2016a, b), and Yıldız and Lekesiz (2017) using gravitational search and hybrid charged system search and Nelder–Mead algorithms, respectively. Hybrid gravitational search and Nelder–Mead algorithm has also been utilized to optimize thin-wall structures (Yildiz et al. 2016; Karagöz and Yıldız 2017). Recently, Kiani and Yildiz (2016) have investigated that the differential evolution algorithm is the best evolutionary optimizer algorithm in vehicle crashworthiness and NVH optimization. Very recently, Pholdee et al. (2017) proposed a hybrid real-code population-based incremental learning and differential evolution algorithm for optimization of an automotive floor-frame structure. Yildiz (2013b) has successfully utilized cuckoo search in selection of optimal machining parameters in milling operations. A multiobjective genetic algorithm is used by Yildiz and Saitou (2011) to obtain Pareto optimal solutions that exhibit trade-offs among stiffness, weight, manufacturability, and assemble-ability. Hereby we would like to mention that in the above discussed study some of the well-known evolutionary and swarm intelligence techniques such as genetic algorithm, particle swarm optimization and cuckoo search have been utilized and their results indicate that there is an enough potential in all these three algorithms if they are hybridized with some effective local search techniques or fine-tuned properly. It also has been noticed during literature review that these three algorithms have good potential to solve security and cryptology problems those are often considered infeasible with traditional approaches (Laskari et al. 2007; Danziger and Henriques 2012; Awad and El-Alfy 2015).

1.2 Motivation

Laskari et al. (2007) have carried out an extensive survey in the domain of cryptology using evolutionary computing and swarm intelligence techniques. Their study motivates us from two aspects. First, cryptographic problems are viewed as discrete optimization tasks and therefore the techniques of evolutionary computation and swarm intelligence family can be utilized to address them. Second, since complex cryptosystems do not reveal any patterns of the encrypted messages or their inner structure, methods from evolutionary computation and swarm intelligence family can constitute a first measure for cryptanalysis (Laskari et al. 2007). Danziger and Henriques (2012) have mentioned that new concepts and ideas have been emerged in CI that can be utilized in cryptology owing to the availability of computational and processing capabilities. Recently, Danziger and Henriques (2012), and Awad and El-Alfy (2015) have investigated that typically, DNA, artificial neural networks and cellular automata techniques are applied to develop new cryptographic systems, and evolutionary computation, swarm intelligence and DNA techniques are applied to perform automated cryptanalysis. In this research, we study swarm intelligence techniques, namely, particle swarm optimization and cuckoo search, and evolutionary computation technique, namely, genetic algorithm due to the following significant reasons:

1. Particle swarm optimization is simple and easy to implement as compared to evolutionary computation techniques (Gonzalez 2007; Lee and Hong 2016).
2. In the case of cuckoo search lesser number of parameters are needed to control as compare to evolutionary computation techniques (Yang and Deb 2009, 2010).
3. Cuckoo search, particle swarm optimization and genetic algorithm are well-studied techniques that have shown good potential in automated cryptanalysis (Laskari et al. 2007; Danziger and Henriques 2012; Awad and El-Alfy 2015). Automated cryptanalysis means the cryptanalytic attacks that run without time-consuming interaction of humans with a search process and finish when the secret key is determined (this is the main advantage of automated attacks). Thus, many cryptanalyst are interested in developing automated cryptanalysis of encryption algorithms. The research in the area of application of metaheuristics in automated cryptanalysis was first reported in 1993 (Spillman et al. 1993; Matthews 1993; Forsyth and Naini 1993). The results have proven that such techniques are highly effective for automated cryptanalysis of reduced cryptosystems.
4. Cuckoo search, particle swarm optimization and genetic algorithm have many similarities as follows: (a) all techniques do not require auxiliary knowledge of the problem. (b) These techniques use population of solutions from the search space which are initially randomly generated. (c) Solutions of same population interact with each other during the search process. (d) These techniques provide more than one solution at the end of search.
5. Recent years have witnessed that the cuckoo search technique is highly effective in the domain of automated cryptanalysis, for example, Bhateja et al. (2015), Jain and Chaudhari (2015a). While, genetic algorithm and particle swarm optimization are extensively used techniques in the domain of automated

cryptanalysis and cryptography; a few remarkable studies have been carried out in the last decade. Nalini and Rao (2007) have demonstrated efficient attacks on the reduced version of DES using genetic algorithm and particle swarm optimization. Ma and Obimbo (2011) have presented cryptanalysis of one-round TEA using evolutionary strategy which is similar to genetic algorithm. Evolutionary strategy has also used by Boryczka and Dworak (2014) for automated cryptanalysis of transposition ciphers. Recently, genetic algorithm and particle swarm optimization have been utilized by Jain and Chaudhari (2014), Bhateja et al. (2015), and Jain and Chaudhari (2015a) in automated cryptanalysis and by Jain and Chaudhari (2015b) in optimizing cryptosystem's non-linear elements.

The above mentioned reasons motivate us to use cuckoo search, particle swarm optimization and genetic algorithm in automated cryptanalysis of the knapsack cryptosystems.

1.3 Related work and our contribution

With regard to the automated cryptanalysis of knapsack cryptosystems, a first remarkable study has been carried out by Spillman (1993) using genetic algorithm (GA), where a small variant of basic MH knapsack cryptosystem has been considered for the attack. Garg and Shastri (2006) have proposed an improved method, namely, IGA (improved genetic algorithm), which is an improvement in the GA method of Spillman. However, Garg and Shastri have proposed the GA attack again on the same small variant. A few more automated attacks of the basic knapsack cryptosystem have been studied in past few years, for example, Abdul-Halim et al. 2008; Muthuregunathan et al. 2009; Palit et al. 2011; Sinha et al. 2011; Jain and Chaudhari 2014. However, the main drawback in the previously studied automated attacks is that none of these studies have provided cryptanalysis of practically representative knapsack cryptosystems. Moreover, all these automated attacks use the Spillman fitness function, i.e., no improvements have been identified in the Spillman fitness function. Therefore, the main objective of this paper is to study and improve the previously proposed GA-based attacks and to introduce some new attacks. In brief, our contributions are as follows:

1. A novel technique based on the cuckoo search, namely, NBCS (novel binary cuckoo search) is proposed especially for automated cryptanalysis (see Sect. 3.3).
2. A simple mutation concept is introduced and appended in NBCS technique in order to improve the search space exploration capability (see Sect. 3.3).
3. Parameters of the cuckoo search is fine-tuned for optimizing the cryptanalysis process (see Sect. 3.3).
4. An improved genotype–phenotype binary particle swarm optimization (IGPBPSO) technique has been developed for automated cryptanalysis (see Sect. 3.4). Further, for a significant comparison, we implement previously proposed GA-based attacks and introduce some improvements, where the new GA algorithm can be referred to as NGA. Following list of improvements have been achieved in order to develop more effective and efficient GA-based attacks.
5. Spillman fitness function has been revised and presented by Jain and Chaudhari (2014) (see Sect. 3.2).
6. Uniform crossover and adaptive mutation operator are developed that make the GA attacks effective (see Sect. 4.1).
7. Crossover and mutation operators are fine-tuned that make the GA attacks efficient (see Sect. 4.1).
8. This paper also presents mathematical model for implementing the complex knapsack cryptosystem. Prime factorization algorithm is also modified for implementing the complex knapsack cryptosystem. Hereby we would like to mention that the proposed model will help researchers to analyze and design the related cryptosystems (for details, see Sect. 2.1.2).

The remainder of the paper is organized as follows: in Sect. 2, we describe the reduced complex knapsack cryptosystem. In Sect. 3, we present automated cryptanalysis of the knapsack cryptosystem using cuckoo search, particle swarm optimization and genetic algorithm. Section 4 present the results and followed by conclusions in Sect. 5.

2 Knapsack cryptosystem considered

This section presents a detailed study of the reduced multiplicative knapsack cryptosystem (RMKC) because of its significance to the proposed metaheuristic based attacks. RMKC is a complex knapsack cryptosystem which is type of a multiplicative MH cryptosystem in which only the knapsack size is reduced, but the length of each element of the knapsack is unchanged (i.e., remains practical sized). This paper considers the reduced knapsack cryptosystem for the systematic demonstration of cryptanalytic attack via new search techniques. Also, it is enviable to consider reduced knapsack cryptosystem which is tractable as well as contains representative features of practical PKC.

Encrypted communication in the case of the general PKCs-first of all, a public key is generated by the receiver (e.g., by Alice) using her own private key. This public key is generated using key generation method. Afterwards, the generated public key is publicized by the receiver. Using publicized public key, the sender (e.g., Bob) encrypts the plaintext and send it over the insecure public

communication channel. Upon receiving the encrypted message (i.e., ciphertext), Alice decrypts the ciphertext using her own private key.

Encrypted communication in the case of the knapsack-based PKC-assuming that the sender has a message $X = (x_1, x_2, \ldots, x_n)$ of n bits length. For sending this message, the sender first determines the ciphertext b using Eq. (1) and then send the generated ciphertext via the public channel. In Eq. (1), $A = (a_1, a_2, \ldots, a_n)$. is a trapdoor knapsack publicized by the receiver as a public key.

$$b = \sum_{i=1}^{n} a_i * x_i \tag{1}$$

It should be noted that the generic knapsack problem is NP-complete, and no polynomial time algorithm is available to solve it. This implies that mounting attacks on the knapsack cryptosystems are difficult. However, from the perspective of a designer, the decryption process should be easy while at the same time it must be difficult for the attacker. In this regard, a possible solution is to convert an easy knapsack sequence into a computationally hard knapsack sequence. The hard knapsack sequence is also referred to as public key or public key vector. The hard knapsack sequence (i.e., public key) can be generated from an easy knapsack sequence in different ways, one of them is presented in the subsequent subsections.

2.1 The RMKC scheme

The RMKC scheme uses a set of n pairwise coprime positive integers $A' = \left(a_1', a_2', \ldots, a_n'\right)$ so that the condition (2) must hold. In the key generation scheme, each secret integer a_i' maps to its exponent using DLP computation. DLP computation is performed in the restricted set of a finite field because no algorithm is known to solve arbitrary DLPs in polynomial time. For instance, consider a multiplicative group Z_p^* of integers over a finite field modulo a prime p and a generator g of Z_p^*. In order to generate key elements DLP computation is employed, where from each element $a' \in Z_p^*$, a can be evaluated as $a = log_g a'$, where $g^a \equiv a' (\mathrm{mod}\, p)$. A small example is helpful here to illustrate the key generation, encryption and decryption process with respect to the RMKC scheme.

Let $A' = (2, 5, 7, 9)$, $m = 961 \left(> \prod_{i=1}^{4} (a_i') \right)$ and base of the logarithm $g = 317$. Compute $A = (183, 290, 421, 14)$, where $\log_{317} 2 \pmod{961} = 183$, $\log_{317} 5 \pmod{961} = 290$, and so on. Consider the plaintext $X = 0101$. The sender obtains the ciphertext b as 83*0 + 290*1 + 421*0 + 14*1 = 304. Afterward, the ciphertext b has sent to the receiver via the public channel. Upon receiving the encrypted code the indented recipient perform the following steps for decrypting the code:

$$\begin{aligned} b' &= 317^{304} (\mathrm{mod} 961) \\ &= 45 \\ &= 2^0 * 5^1 * 7^0 * 9^1 \end{aligned}$$

In this way, the recipient recovers the plaintext: 0101.

$$m > \prod_{i=1}^{n} \left(a_i'\right) \tag{2}$$

2.1.1 Choosing size of pameters

If 40 random pairwise coprime integers each of typically 100-bits long are taken, then due to condition (2) approximately 4000-bits are required to represent m. In that case, if first 40 prime numbers are taken then m requires $\log_2 m$ bits, i.e., 227-bits. Consequently, each public key element typically requires 227-bits. This parameter selection strategy will be secure as well as will maintain the encryption speed.

2.1.2 Implementation detail: key generation

Recall from Sect. 2.1, the ciphertext is obtained as the sum of some of the exponents which are then easily converted into a product of corresponding secret integers a_i', and thereby decrypt the original text. Though this process makes the decryption stage easy, the challenge is to map secret integers a_i' into their exponents, i.e., implementation of DLP computation over a finite field. Implementation of DLP computation in a large finite field is very hard in general. However, the Pohlig–Hellman algorithm (Pohlig and Hellman 1978) is efficient in practice for some special cases, e.g., a multiplicative group Z_m^* of integers over a finite field modulo a prime m. Z_m^* is said to be a cyclic group if Z_m^* has a generator (see Definition 3). But, if m is prime then Z_m^* must have a generator, hence, it is said to be a cyclic group. Consequently, the following are challenges in implementing key generation scheme of RMKC using Pohlig–Hellman algorithm:

1. To find the prime factorization of $p = p_1^{e1} p_1^{e2} \ldots p_1^{er}$, where p is the order of Z_m^*(see Definition 1).
2. To find the generator of cyclic group Z_m^*
3. As an intermediate result, to find the order of a group element.

Consider, a cyclic group Z_m^*, where m is a 227-bit prime number (reason has been discussed in the above section). Since this group must have first 40 prime numbers, thus we focus on efficiently mapping the prime numbers.

Algorithm 1: Determining prime factorization

Data: a composite integer p that is not a prime power
Result: prime factorization of p as ${p_1}^{e_1}, {p_2}^{e_2}, \ldots, {p_r}^{e_r}$
Procedure PrimeFactorization (p)
Set $x \leftarrow 2$, $y \leftarrow 2$, $z \leftarrow 1$
Set constants $c_1 \leftarrow 169$, $c_2 \leftarrow 4$, $c_3 \leftarrow 22$
repeat
  if $z \neq p$ && $z \neq c_1$ **then**
    Compute $x \leftarrow x^2+1 \pmod p$
    Compute $y \leftarrow y^2+1 \pmod p$
    Compute $y \leftarrow y^2+1 \pmod p$
  end if
  if $z = p$ && $p \pmod{c_2} \neq 0$ **then**
    Compute $x \leftarrow x^2+2 \pmod p$
    Compute $y \leftarrow y^2+2 \pmod p$
    Compute $y \leftarrow y^2+2 \pmod p$
  end if
  if $z = p$ && $p \pmod{c_2} = 0 \,\|\, z = c_3$ **then**
    Compute $x \leftarrow x^2+3 \pmod p$
    Compute $y \leftarrow y^2+3 \pmod p$
    Compute $y \leftarrow y^2+2 \pmod p$
  end if
  if $z = c_1$ **then**
    Compute $x \leftarrow x^2+5 \pmod p$
    Compute $y \leftarrow y^2+3 \pmod p$
    Compute $y \leftarrow y^2+2 \pmod p$
  end if
    Compute $z \leftarrow \text{GCD}(x\text{-}y, p)$
  if $z > 1$ && $z < p$ **then**
    if z is probable prime with probability > 0.5 **then**
      Append z in the list of prime factorization
    else
      return CALL PrimeFactorization(z)
    end if
    Set $z \leftarrow \frac{p}{z}$
    if z is probable prime with probability > 0.5 **then**
      Append z in the list of prime factorization
    else
      return CALL PrimeFactorization(z)
    end if
    return 0
  end if
  if $z = 1$ **then**
    Continue LOOP
  else if $z = p$ **then**
    Terminate the algorithm with failure
      end if
  end if
until (false)

That is, secret integers a_i' to their unique exponents that forms a public key of size 40. In this regard, we propose an efficient implementation model which is shown in Fig. 1 that fulfils the above-mentioned challenges efficiently. This

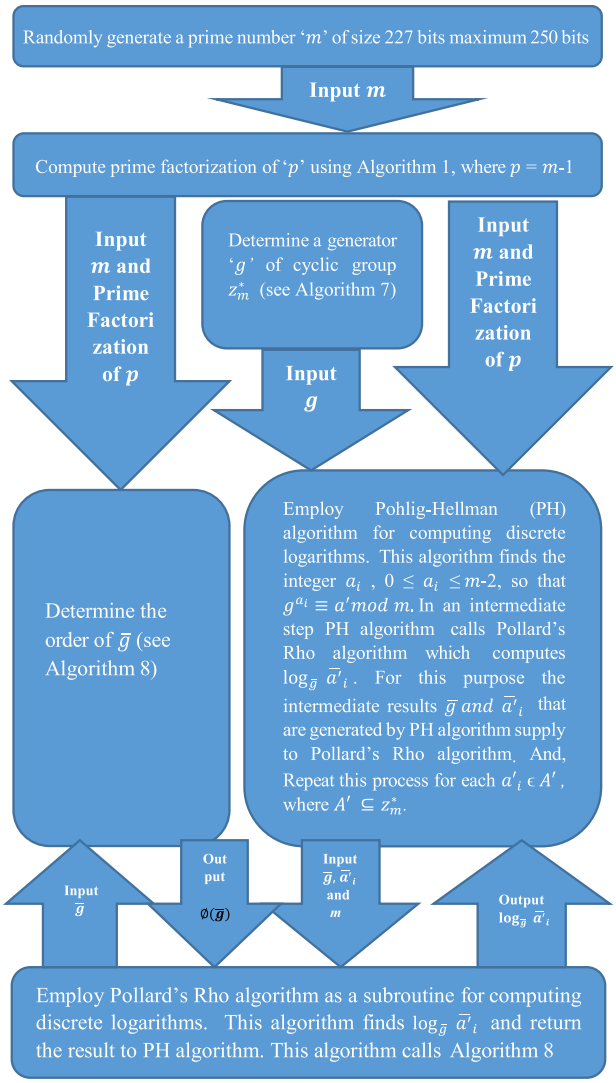

Fig. 1 Implementation model: key generation in RMKC scheme

model presents the order of execution of various algorithms that are used to generate a public key. The Pohlig–Hellman algorithm is employed for computing discrete logarithms that takes base g, i.e., a generator of Z_m, integer a_i', modular m and prime factorization of p as inputs and generate exponent a_i as output, where $p = m - 1$ and $1 \le i \le 40$.

For the purpose of determining prime factorization of p which is a 227-bit integer number, the Pollard Rho integer factorization algorithm is modified according to our requirements. The corresponding pseudocode is shown in Algorithm 1. Pseudocode for finding the generator of a cyclic group is shown in Algorithm 7 (see Appendix A). As a subroutine, the Pohlig–Hellman algorithm executes the Pollard Rho algorithm that computes discrete logarithms. Subsequently, the Pollard Rho algorithm executes Algorithm 8 (see Appendix A) that finds the order of the element of a multiplicative group Z_m and compute discrete

logarithm and finally returns the result back to the Pohlig–Hellman algorithm. Basic concepts about the above-mentioned algorithms are extensively treated by Menezes et al. (2010).

Definition 1 $Z_q = \{0, 1, 2, \ldots, q-1\}$ is a set of integers modulo q. Each integer $c \in Z_q$ is included in Z_q^*, if $gcd(c, q) = 1$. 1. In fact, if q is prime, then in $Z_q^*, = \{c | 1 \leq c \leq q-1\}$. The order of Z_q^* can be represented as $\emptyset(q)$ (see *Definition* 2), where $\emptyset(q) = \left|Z_q^*\right|$.

Definition 2 (*Euler totient function*) The function $\emptyset$ is called the Euler totient function or Euler phi function and for $q \geq 1, \emptyset(q)$ denote the number of integers in the interval [1, q] those are relative prime to q. Now, it is important to consider the properties of $\emptyset$ which are as follows.

1. If q is prime, then $\emptyset(q) = q - 1$.
2. $\emptyset$ is multiplicative. That is, if $gcd(x, y) = 1$, then $\emptyset(xy) = \emptyset(x).\emptyset(y)$.
3. If $q_1^{e1} q_2^{e2} \ldots q_r^{er}$ is the prime factorization of q, then

$$\emptyset(q) = q\left(1 - \frac{1}{q_1}\right)\left(1 - \frac{1}{q_2}\right)\ldots\left(1 - \frac{1}{q_r}\right)$$

Definition 3 Consider $g \in Z_q^*$. If $\emptyset(g) = \emptyset(q)$ is then g is said to be a generator of Z_q^*. Following this reasoning, if Z_q^* has a generator, then Z_q^* is said to be cyclic.

3 Utilization of cuckoo search, particle swarm optimization and genetic algorithm in automated cryptanalysis of the considered knapsack cryptosystem

3.1 Preliminaries: cuckoo search, particle swarm optimization and genetic algorithm

Cuckoo search (CS) is one of the recent search heuristic that has been proposed by Yang and Deb (2009). This metaheuristic has formed by inspiring from the obligate brood parasitic behaviour of few cuckoo species in combination with Lévy flights behaviour of some birds and fruit flies (Yang and Deb 2009, 2010). Algorithm 2 shows the standard template of the CS technique, where from an existing nest $x_j(t)$, a new nest $x_j(t+1)$ is generated via Lévy flight as (Yang and Deb 2009; Yang et al. 2014):

Algorithm 2: Standard Cuckoo Search (Yang & Deb, 2009)
Generate initial population of m host nests x_i, where$i = 1, 2, \ldots, m$.
repeat
1. Select a cuckoo (i.e., candidate solution, say, x_j) randomly, and modify that solution to generate new solution by Levy flights
2. Compute the cost of new solution, let it be f_j
3. Randomly choose a nest among m host nests, say x_k
4. if ($f_i < f_k$) comment: let, the problem has minimization objective
5. Replace x_k by the new solution x_j
6. Abandon a fraction (P_a) of worst nests/solutions and construct new ones
7. Keep the best solutions
8. Rank the solutions and find the current best

until (Termination condition is satisfied)
Post-process results

$$x_j(t+1) = x_j(t) + \mu * l \tag{3}$$

Lévy flights is an effective random walk approach, which is achieved using Eq. (3). Random walk is basically a Markov chain in which subsequent position is decided using the current position (i.e., first term in Eq. (3)) and the probability of transition (i.e., second term in Eq. (3)) Yang and Deb (2009). The probability of transition is given by $\mu^* l$ where $\mu > 0$, is a step-size escalating factor. The step-size is represented by term l in Eq. (3), which is taken from a Lévy stable distribution (Mantegna 1994). Theoretically, random walk using Lévy flights is an effective method for the search space exploration because the distribution of its step-size is pseudo-random. Nevertheless, generation of pseudo-random steps that accurately follow the Lévy stable distribution is not an easy task. In view of implementation, one of the straightforward and most efficient method is Mantegna's (1994) algorithm that generates a stochastic variable (Yang 2014). This stochastic variable has probability density which is close to Levy stable distribution characterized by an arbitrarily chosen control parameter ($0.3 \leq \lambda \leq 1.99$). The step-size l can be calculated as (Yang 2014):

$$l = \frac{u}{v^{1/\lambda}} \tag{4}$$

In Eq. (4) u and v are two Gaussian stochastic variables with a zero mean, and standard deviations of σ_u and σ_v, respectively; σ_u and σ_v can be given as (Yang 2014):

$$\sigma_u(\lambda) = \left[\frac{\Gamma(1+\lambda)\sin(\pi\lambda/2)}{\Gamma((1+\lambda)/2)\lambda 2^{(\lambda-1)/2)}}\right]^{1/\lambda} = 0.696575 \text{ and } \sigma_v(\lambda) = 1, if \ \lambda = 1.5, \text{ where } \Gamma(z) = \int_0^\infty t^{z-1}e^{-z}dt \tag{5}$$

Particle swarm optimization (PSO) is a swarm intelligence technique which has been originated via simulation studies of birds flocking. In literature, Kennedy et al. (1995), and Kennedy and Eberhart (1997) introduced two different version of PSO that are known as real-PSO and binary-PSO (BPSO), respectively. Since this paper utilizes a BPSO technique, thus in further discussion, BPSO is described in detail.

BPSO method is initialized by a random population of particles. The velocity, position, and personal-best position vectors denoted by $V_i = (v_{i1}, v_{i2}, \ldots, v_{in}), X_i = (x_{i1}, x_{i2}, \ldots, x_{in})$, and $P_i = (p_{i1}, p_{i2}, \ldots, p_{in})$ are associated with each particle, where x_{ij} and $p_{ij} \in (0, 1)$; v_{ij} is the velocity of the jth element of the ith particle constrained by v_{max} (maximum velocity allowed).

In the above mentioned vectors, $i = 1, 2, \ldots, m$ (where $m$represents number of particles) and $j = 1, 2, \ldots, n$ (where n represents each particle is a potential solution in the n dimensional space). For optimization of the solutions, a vector $G = (g_1, g_2, \ldots, g_n)$ is generated for velocity update, where G is the global best position (*gbest*) for global topology and local best position (*lbest*) for local topology models. Using $P_i G$ and current position X_i, the next velocity of ith particle is computed using Eq. (6):

$$v_{ij}(t+1) = w * v_{ij}(t) + c_1 r_1\left(p_{ij}(t) - x_{ij}(t)\right) + c_2 r_2\left(g_j(t) - x_{ij}(t)\right) \tag{6}$$

In Eq. (6), w is the inertia weight that handles exploration and exploitation abilities of the particles (Shi and Eberhart 1998). A high value (e.g., 0.9) of w encourages global exploration and a low value (e.g., 0.1) provides a local exploitation (Shi and Eberhart 1998). In our experiments, w has been fine-tuned, and the details have been discussed in Sect. 4.1. c_1 and c_2 are acceleration constants that have been set to the standard value 2.05 (Engelbrecht 2007). r_1 and r_2 are random variables in the interval [0, 1] obtained from a uniform distribution $U(0, 1)$. $v_{ij}(t)$ and $v_{ij}(t+1)$ represents the current velocity and next velocity, respectively of the jth element of the ith particle. Using next velocity, next position of the ith particle is computed using Eqs. (7) and (8):

$$x_{ij}(t+1) = \left\{ \begin{array}{ll} 0, & if\ U(0,1) \geq Sig\left(v_{ij}(t+1)\right) \\ 1, & if\ U(0,1) < Sig\left(v_{ij}(t+1)\right) \end{array} \right\} \tag{7}$$

$$Sig\left(v_{ij}(t+1)\right) = \frac{1}{1+e^{-v_{ij(t+1)}}} \tag{8}$$

where $Sig\left(v_{ij}(t+1)\right)$ is a sigmoid function that is used to transform velocity in the interval [0, 1].

Genetic algorithm (GA) is one of the most popular evolutionary algorithms that has been emerged based on the concept of imitating the evolution of a species (Goldberg and Holland 1988; Michalewicz 2013). In genetic algorithm, a population of individuals (or chromosomes) is generated using an intelligent method or a random method (Goldberg and Holland 1988; Srinivas and Patnaik 1994; Gonzalez 2007; Michalewicz 2013). Each of these individuals encodes as a binary string that represents a possible candidate solution to the problem at hand. In each iteration the survival strength of each candidate solution is measured by a fitness function (Goldberg and Holland 1988; Srinivas and Patnaik 1994; Gonzalez 2007; Michalewicz 2013). The evolutionary process constrained by three genetic operators: selection, crossover, and mutation. Through selection procedure, individuals are selected that enters to the crossover process. The crossover operator alters two or more parents to create offspring; a probabilistic crossover rate is usually used to generate offspring (Goldberg and Holland 1988; Srinivas and Patnaik 1994; Vose 1999; Gonzalez 2007; Michalewicz 2013). Mutation operator produces one child from one parent by flipping a bit/bits of the parent; a probabilistic mutation rate is usually used that determines whether a particular change is to be occurred within an individual or not (Goldberg and Holland 1988; Srinivas and Patnaik 1994; Vose 1999; Gonzalez 2007; Michalewicz 2013).

3.2 Fitness function

CS, PSO and GA are population-based search algorithms that initiate with a population of individuals (particles in the case of PSO, and host nests in the case of CS), where individuals are generated, typically, randomly. Each of these individuals represents a possible candidate solution to the problem at hand. After each iteration, each individual solution is tested to check whether a required solution is found; if the solution is not found, our approach is to remove the 'k' worst solutions and to breed the 'k' best solutions. An efficient way of testing each outcome solution is to assign a figure of merit that indicates how close the solution is meeting the overall constraints. This can be achieved by applying the fitness function to the test. In a nutshell, fitness function (or objective function) is an important component of the CI methods that assist them in finding the best solutions. In the next paragraph, the fitness function is described in detail. When we run the meta-heuristic attacks, at each iteration, we get a set of binary strings where each string represents a feasible solution (i.e., a candidate plaintext). In the case of cryptanalysis, we

focus on designing a fitness function that assists meta-heuristics in finding the exact plaintext from the set of candidate plaintexts by assigning them a number ≤ 1. Since the attacker can only reveal the public key and the ciphertext, a fitness function can only be designed using *TotalSum* (sum of all elements of the public key) and the *Target*(ciphertext). Following this reasoning, Jain and Chaudhari (2014) have revised the fitness function that was previously used for the purpose of the cryptanalytic attack by Spillman (1993), Garg and Shastri (2006), AbdulHalim et al. (2008), Muthuregunathan et al. (2009), Palit et al. (2011), Sinha et al. (2011). Details about the fitness function is as follows: We have analysed the fitness function that was introduced by Spillman and realize that the concept of the normal probability distribution can be employed to design a better fitness function (the problem is considered to be of maximization type for which the maximum fitness value can be 1). The main concern in the design of a fitness function is at what amount the term *Difference* should be normalized so that it best describes the fitness of investigated Sum to the expected *Target*. After several experimental runs on relevant data, we have found that the power term to be used over *Difference* should be either 0.5 or 0.6 to normalize the difference between *Target* and *Sum* appropriately. All possible fitness functions have been tested by substituting power term in a range of 0.1–1 with step size 0.1, but the normally distributed curves have been obtained with power term 0.5 and 0.6 (see Fig. 2). In this paper, we choose both fitness functions in adaptive way. That is when evolution starts Eq. (11) is used to measure the quality of the investigated solutions. However, after some iterations (for instance, 100) if no improvements in the quality of solutions is found Eq. (12) is employed and vice versa.

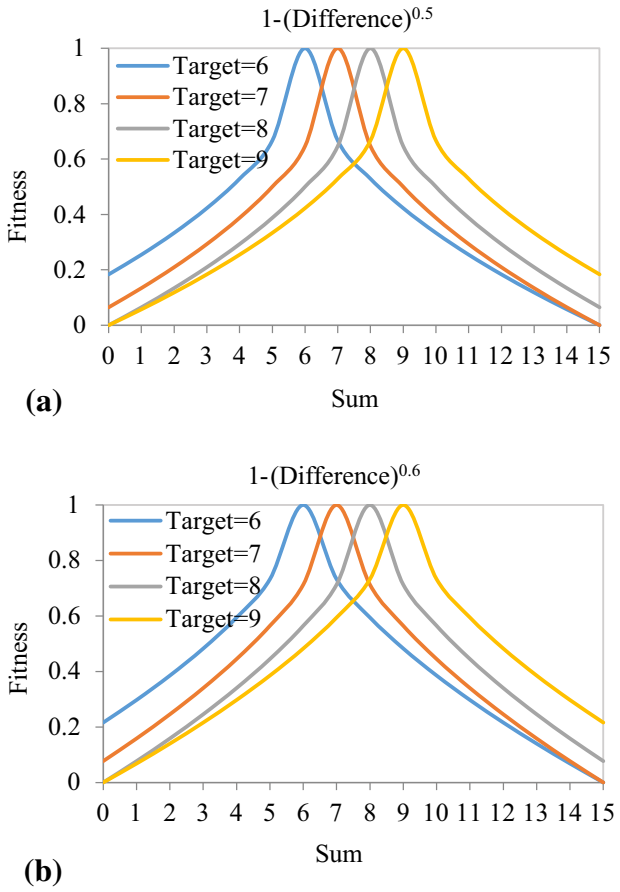

Fig. 2 Analysis of fitness function

Spillman fitness function

$$\text{if}(Sum < Target), Fitness = 1 - (DifferenceLess)^{\frac{1}{2}}, \quad (9)$$

$$\text{otherwise}, Fitness = 1 - (DifferenceLess)^{\frac{1}{6}}, \quad (10)$$

The fitness function used in this research adaptively are as follows:

$$Fitness = 1 - (Difference)^{0.5}, \quad (11)$$

$$Fitness = 1 - (Difference)^{0.6}, \quad (12)$$

where

$$DifferenceLess = (|Target - Sum|)/Target)$$

$$Difference = (|Target - Sum|)/max\ (TotalSum - Target, Target).$$

3.3 Automated cryptanalysis using NBCS

Algorithm 3 shows the pseudocode of the NBCS algorithm. The input to the algorithm is the ciphertext b and the public key vector A. The task of the algorithm is to recover the plaintext from the known ciphertext. The algorithm works as follows: initially, a population of host nests is generated randomly, where each of the nests is encoded as a binary string that represents a candidate plaintext. For each iteration, the survival strength of each candidate plaintext is measured by the adaptive fitness function. From the population of candidate plaintexts, a plaintext of highest fitness is selected which is called as best plaintext. Also a random plaintext is selected from the population of candidate plaintexts which is called as ith plaintext. Using ith plaintext, a new plaintext is constructed via Lévy flights, i.e., using Eq. (3). The problem is that the new vector which is generated by Eq. (3) will belong to the real space. The importance of Eq. (3) is that it builds a new plaintext from the existing binary plaintext via Lévy flights which is an efficient approach, since the step-size is heavy tailed and any large step is possible. That is the existing plaintext can be converged into the exact plaintext. Hence, we do not change this equation, rather we transform the new plaintext from real space to binary space using Eqs. (13) and (14).

Algorithm 3: Pseudocode for Effective Binary Cuckoo Search Algorithm for Automated Cryptanalysis

1. **Initialization:** Generate the initial binary population of m host nests randomly, where each host nest represents a candidate plaintext. Call this population $CPlaintexts$.
2. **Repeat**
3. Compute the cost of each of the plaintexts of the $CPlaintexts$ using Eq.(8) or Eq. (9) adaptively.
4. Select a plaintext from $CPlaintexts$ that has the highest cost, call this plaintext $BestPlaintext$.
5. Select a plaintext randomly (say, $CPlaintexts_i$), and generate a new solution (abbreviates as $NewPlaintext$ using $CPlaintexts_i$ via Levy flights as: $NewPlaintext_j$=$CPlaintexts_{ij}$+μl, where $1 \leq j \leq n$, $\mu = 0.01$ and l is computed using Eq. (4) with $\lambda = 1.5$.
6. Convert $NewPlaintext$ in the binary form using Eqs. (10) and (11).
7. Apply simple mutation (see Algorithm 4) on the $BestPlaintext$.
8. Compute the cost of $NewPlaintext$ using Eq. (8) or Eq. (9) adaptively. If the cost of $NewPlaintext$ is better (i.e., higher) than $BestPlaintext$ then it becomes the new $BestPlaintext$. If it is then go to Step 7.
9. Abandon a fraction p_a of worst plaintexts. Here, we create new plaintexts from the best plaintexts as replacement of abandon plaintexts by applying the simple perturbation mechanism. According to the perturbation mechanism, two randomly selected elements of the chosen plaintext is interchanged. The value of p_a is settled to 0.05 based on the extensive experiments.
10. **Until** Maximum Iterations (or) Fitness=1 is true.
11. Output the best solution from the $CPlaintexts$.

Algorithm 4: Pseudocode for Simple Mutation

1. P_m is the probability of mutation which is limited to 0.02. This limit is decided based on the experiments.
2. **Initialize** $j = 1$
3. **Repeat**
4. **if** $U[0,1] < P_m$ then
5. if $NewPlaintext_j$= 0 then
6. $NewPlaintext_j$= 1
7. else
8. $NewPlaintext_j$= 0
9. **Until** $(j > n)$

$$Sig\left(NewPlaintext_j\right) = \frac{1}{1 + e^{-NewPlaintext_j}} \tag{13}$$

$$NewPlaintext_j = \left\{ \begin{array}{ll} 1, & if\ U(0,1) < Sig\left(NewPlaintext_j\right) \\ 0, & if\ U(0,1) \geq Sig\left(NewPlaintext_j\right) \end{array} \right\} \tag{14}$$

The above equations perform the following two operations for transformation:

1. Bring the solution in the interval [0, 1] using the sigmoid function which can be represented as $Sig(Nnewplaintext_j)$.
2. Generate a random number in the range [0, 1]. If the random number is lower than the $Sig\left(NewPlaintext_j\right)$ then $Nnewplaintext_j$ takes the value 1. Otherwise, $Nnewplaintext_j$ takes the value 0.

The problem with sigmoid (or probability) function is high dissimilarity in distribution of 1 and 0. For bigger $NewPlaintext_j$ values the probability of getting 1 will decrease and probability of getting 0 will increase, and vice versa, i.e., bigger values of $Newplaintext_j$ results in low exploration (for instance, for $Newplaintext_j = 5.0$, there is a small probability of 0.0067 that a bit will be 1) Bansal and Deep (2012). Thus in order to balance the performance of exploration in cuckoo search algorithm, a simple mutation operator is used which is shown in Algorithm 4. Afterward, the cost of new plaintext is evaluated. If the cost of new plaintext is better than the best plaintext, then it became the best plaintext, and again the new plaintext is generated using the same process discussed above. Otherwise, 0.05 fractions of the plaintexts that have the low fitness are replaced. For replacement, two randomly selected elements of the selected worst plaintext is interchanged, in this way a new plaintext is inserted in the population. The above process is repeated till the termination condition is reached. Since it is not always possible to recover the exact plaintext, the termination condition of the algorithm is set to a fixed number of generations or the best fitness (i.e., '1') is achieved. The output of the algorithm is either an exact plaintext (i.e., a single nest) or a set of best n plaintexts (i.e., nests with fitness nearly '1').

3.4 Automated cryptanalysis via IGPBPSO

BPSO technique is simple and easy to implement (Lee and Hong 2016). However, the original BPSO method introduced by Kennedy and Eberhart (1997) has problem of early convergence and weakness of global search capability, therefore, the original BPSO method being stuck in local optima (Lee and Hong 2016). In order to skip from local optima various improved versions of original BPSO method have been proposed in the literature (for instance, Pampara et al. 2005; Sadri and Suen 2006; Khanesar et al. 2007; Bansal and Deep 2012; Lee and Hong 2016). Recently, Lee and Hong (2016) have significantly improved the original BPSO method using genotype–phenotype concept of genetic algorithm. This modified method can be termed as genotype–phenotype binary particle swarm optimization (GPBPSO). As a case study, Lee and Hong (2016) have successfully applied the GPBPSO technique for solving the multidimensional knapsack

problems. Unlike real-PSO, in BPSO the current particle position is not used for determining next particle position. This drawback has been resolved in GPBPSO.

Instead of one particle position, two particle positions, namely genotype $x^g$and phenotype x^p particle positions have been introduced by Lee and Hong (2016). In the proposed IGPBPSO technique, the genotype updating equation is similar to the position update equation used in real-PSO (see Eq. (16)). The next genotype position is passed in the sigmoid function for computing the phenotype position (see Eqs. (17) and (18)). For velocity update, phenotype position is used (see Eq. (15)). In Eq. (15), we choose G as the global best position $gbest$, because recent results show that as compared to local topology (e.g., Von Neumann) global topology provides more optimal solution (Jain and Bharadwaj 2017).

$$v_{ij}(t+1) = w * v_{ij}(t) + c_1 r_1 \left(p_{ij}(t) - x^p_{ij}(t)\right) + c_2 r_2 \left(g_{bestj}(t) - x^p_{ij}(t)\right), \tag{15}$$

$$x^g_{ij}(t+1) = x^g_{ij}(t) + v_{ij}(t+1), \tag{16}$$

$$x^p_{ij}(t+1) = \left\{ \begin{array}{ll} 0, & if\ U(0,1) \geq Sig\left(x^g_{ij}(t+1)\right) \\ 1, & if\ U(0,1) < Sig\left(x^g_{ij}(t+1)\right) \end{array} \right\}, \tag{17}$$

where

$$Sig\left(x^g_{ij}(t+1)\right) = \frac{1}{1+e^{-x^g_{ij}(t+1)}}. \tag{18}$$

Algorithm 5 shows pseudocode for IGBPSO. Input to the algorithm is the ciphertext b and the public key vector A. The task of the algorithm is to recover the plaintext corresponding to the known ciphertext. Recall that two vectors, namely, particle position vector and particle velocity vector are associated with each particle, where both vectors are represented as binary arrays of length n (n is the size of the public key). Algorithm 5 works as follows: after various initializations (see Algorithm 5, Steps 3–7), the algorithm progresses iteratively. Upon each iteration, the proposed fitness function measures the position of ith particle (i.e., ith candidate plaintext) and updates the corresponding $Lbest_i$ (see Algorithm 5, Step 9–14). The fitness function is also used to measure and update the $Gbest$ (see Algorithm 5, Step 15). Finally, velocity and position of each particle are updated using Eqs. (15) and (17), respectively.

Note that BPSO-based techniques have a high probability to find the optimal solution by limiting the value of v_{max} and w. In order to achieve a high success rate, the selection of suitable values of v_{max} and w are extremely important. We choose these values following the guidelines in (Shi and Eberhart 1998; Khanesar et al. 2007; Bansal and Deep 2012; Lee and Hong 2016) and through extensive testing. In the experiments, we limit the value of $v_{max} = 1$(see Sect. 3.1). In the case of w, a time-dependent inertia weight has been chosen which is initialized with 0.9 and decreased linearly after each iteration using Eq. (19). In starting 60% of total iterations, w is decreased with delta (= 0.5/MaxIterations) and then for remaining iterations it is kept constant.

$$w = \frac{w(in) - w(fin) * (MI - INum(t))}{MI} + w(fin) \tag{19}$$

where $w(in) = 0.9$, $w(fin) = 0.4$, MI : Number of Maximum Iterations, $INum(t)$ Iteration number at time t.

Algorithm 5: Pseudocode of IGPBPSO for Cryptanalysis

1. **Input**: public key vector and the ciphertext
2. **Output**: plaintext corresponding to the ciphertext
3. **for** $i = 1, 2, \ldots, m$ do
4. Initialization of m particles: randomly initialize the position vector X_i and the velocity vector V_i that are associated with the ith particle, where each vector is n-dimensional.
5. Initialize P_i by corresponding X_i (i.e., by a candidate plaintext)
6. **end for**
7. Initialize G by the particle position vector, say X_j, that have the best fitness value among all the particles (i.e., among all the candidate plaintext)
8. **repeat**
9. **for** each particle i=1,2,…, m **do**
10. Compute fitness of X_i using Eq. (8) or Eq. (9) adaptively
11. **if** the fitness of X_i is better than the fitness of P_i
12. (the best fitness value in history corresponding to ith
13. particle) then
14. Update P_i by the current X_i
15. **end if**
16. **end for**
17. **Update** G, for this iteration by the particle position vector, say X_j, that have the best fitness value among all the particles
18. **for** each particle i=1,2,…, m do
19. Using Eq. (16), update particle velocity
20. Using Eq. (17), update genotype position.
21. Compute phenotype position using Eqs. (18) and (19).
22. **end for**
23. **until** (Maximum Iterations (or) Fitness=1 is true)

3.5 Automated cryptanalysis via NGA

Algorithm 6 shows the pseudocode of the NGA algorithm. The input to the algorithm is the ciphertext b and the public

key vector A. The task of the algorithm is to recover the plaintext from the known ciphertext via evolutionary processes constrained by three genetic operators: selection, crossover, and mutation. The algorithm works as follows: initially, a population of individuals (chromosomes) is generated randomly, where each of the individuals is encoded as a binary string that represents a candidate plaintext corresponding to the known ciphertext. For each iteration, the survival strength of each candidate plaintext is measured by the adaptive fitness function. During experiments, the selection pressure can be adjusted easily by changing the tournament size, therefore, the tournament selection procedure is adopted. The selection procedure runs several tournaments among few individuals, where individuals are selected from the population using a uniform random process. Afterward, we choose the winner of the tournament for crossover (the one with the best fitness). A new population is created by applying uniform crossover to each pair of selected individuals (for details on crossover and its fine-tuning, see Sect. 4.1). Finally, a mutation operator is applied to every individual that flips a bit (or bits) at random positions (for details on mutation and its fine-tuning, see Sect. 4.1). Since it is not always possible to recover the exact plaintext, the termination condition of the algorithm is set to a fixed number of generations or the best fitness (i.e., '1') is achieved. The output of the algorithm is either an exact plaintext (i.e., a single chromosome) or a set of best plaintexts (i.e., chromosomes with fitness nearly '1').

Algorithm 6: Pseudocode of NGA

1. **Input:** public key vector and ciphertext.
2. **Output:** plaintext corresponding to ciphertext
3. **Initialization:** a random population of chromosomes is generated.
4. **repeat**
5. Evaluate the fitness of each chromosome using Eq. (8) or Eq. (9) adaptively.
6. Select the best fitness chromosomes using tournament selection.
7. Apply uniform crossover with probability of 0.3 to each pair of selected chromosomes.
8. Apply low probability mutation M2 and M3 (see Section 4.1) to candidates found in the previous step.
9. Next generation population is scanned to update a list of best chromosomes.
10. **until** (Maximum Iterations (or) Fitness=1 is true)

4 Results

Encryption scheme RMKC and cryptanalysis algorithms GA, IGA, NGA, IGPBPSO and NBCS have been implemented in Java 2.0 on Intel Quad-Core processor i7-2600 CPU (@3.40 Ghz) with 32 GB RAM. All attack algorithms were run and tested on the considered encryption scheme. The performance of each algorithm is measured in terms of a number of candidate plaintexts are examined (i.e., searched space), success rate (i.e., number of times the exact plaintext has been recovered), and the time taken by the algorithm.

4.1 Fine tuning of crossover and mutation operator for improving GA-based attacks

Rather than traditional crossover operator used by various researchers: Spillman (1993), Garg and Shastri (2006) and Muthuregunathan et al. (2009), Palit et al. (2011), Sinha et al. (2011), we use uniform crossover operator. Unlike traditional crossover operators (single-point crossover and two-point crossover), uniform crossover enables the parent chromosomes to contribute at the gene level rather than the segment level. Uniform crossover means a mixing of genes between two parents at multiple chosen random crossover points with a fixed ratio. Actually, this scheme makes every place a potential crossover point. Through extensive experiments, the fixed mixing ratio is fine-tuned to 0.3, because this ratio generates best results for the case considered. Since the mixing ratio is limited to 0.3, the offspring has approximately 30% of the genes from one parent and the remaining 70% from another parent, where crossover points are chosen randomly. After evolving the new population through crossover, each individual of the new population is mutated with a low probability that prevents the algorithm from being stuck in a local optimum. We have tested the following three different mutation operators on the RMKC scheme: (1) a simple random mutation operator with low probability. (2) a mutation operator that was used by Spillman in his study, and (3) adaptive mutation operator. The following results have been observed: in the case of the M1 mutation operator, the convergence of NGA algorithm is fast, but the amount of state space searched is much higher than the remaining M2 and M3 operators. The goal of this study is to develop an attack algorithm that can provide higher success rate, while the state space searched should be minimum. In this regard, we finalize the mutation operator that has been proposed by Spillman (M2) and the adaptive mutation operator (M3) proposed in this paper, because these operators' generate better results with respect to the state space searched and success probability. However, during execution of the NGA algorithm, one mutation operator between M2 and M3 is selected through a uniform random process.

1. **First mutation operator (M1)** For each of the individual, few bits are randomly mutated with low probability (0.02).

2. **Second mutation operator (M2)** For each of the individual, half of the time bits are randomly mutated with low probability (0.05). Another half of the time bits is being swapped with its neighbour; this is also with the low probability of 0.05. Along with this operation, the order of a set of bits is inverted, where the set is chosen between two random points.
3. **Third mutation operator (M3)** This adaptive mutation starts with very low probability (0.015). However, the rate of mutation is increased when the evolution starts to languish.

4.2 Cryptanalytic results obtained via GA-based algorithms: GA, IGA, and NGA

Recall from Sects. 3.2 and 4.1, the main difference among GA, IGA and NGA algorithms are in the design of the fitness function, choice of the selection mechanism, design of crossover and mutation operators and the fine tuning of crossover and mutation operators. Consequently, the aim of this section is to investigate the best GA algorithm among three GA schemes experimentally so that a fair comparison can be obtained between NBCS, IGPBPSO and the best of three GA algorithms. For this purpose, the RMKC scheme with knapsack size 10 is attacked using all three GA-based algorithms. For determining the best GA algorithm among three, we conduct experiments based on the following three criteria with results:

1. Number of plaintexts examined for successful attacks (maximum $= 2^{10} = 1024$). For this criteria, each algorithm has been tested 50 times on a single random ciphertext, and the statistics of the results are shown in Tables 1 and 2, and Fig. 3. From Table 1, it is observed that the minimum and the maximum number of plaintexts examined by NGA algorithm are only 214 and 467, respectively that are lesser comparatively to GA and IGA algorithms. From Table 2, it is observed that the average number of plaintexts examined by NGA algorithm are only 322.68, which is much lesser than 376.42 of IGA algorithm and 390.18 of GA algorithm. Also, the standard deviation is lower and more stable in the case of NGA algorithm. From Tables 1 and 2 and Fig. 3, we conclude that with respect to the criteria of the number of plaintexts examined, NGA algorithm outperforms GA and IGA algorithms.
2. Number of iterations required for successful convergence. For this criteria, each algorithm has been tested 50 times on a single random ciphertext, and the statistics of the results are shown in Tables 3 and 4, and Fig. 4. From Table 3, it is observed that the minimum and the maximum number of iterations required by NGA algorithm for successful convergence are 4 and 15,

Table 1 Number of plaintexts examined for successful attacks, fifty independent runs of GA, IGA and NGA algorithms. Data have been arranged in ascending order to indicate smallest (Min), Q1, Q2 (Median), Q3 and Largest (Max) values

GA	IGA	NGA	Remark	GA	IGA	NGA	Remark
235	223	214	Min	386	367	324	
239	225	221		398	375	327	
250	226	235		407	387	328	
255	228	246		417	394	331	
261	235	253		421	407	332	
263	242	257		432	411	332	
268	249	258		452	419	333	
275	254	265		455	427	334	
284	259	273		458	435	335	
286	263	275		468	448	335	
287	275	279		473	468	336	
288	286	281	Q1	481	476	337	Q3
292	289	284		491	489	343	
293	294	285		503	503	351	
299	299	287		508	505	359	
303	302	287		511	507	367	
305	304	288		514	509	379	
306	307	288		519	511	392	
307	309	291		526	512	415	
321	314	293		531	517	423	
333	331	295		538	519	431	
347	336	296		545	526	436	
354	341	299		549	534	438	
372	349	321		557	539	456	
378	353	322	Med.	568	543	467	Max

Table 2 Mean and standard deviation data corresponding to Table 1 for successful attacks

Method →	GA	IGA	NGA
Mean	390.18	376.42	322.68
Standard Deviation	104.56	104.68	60.22

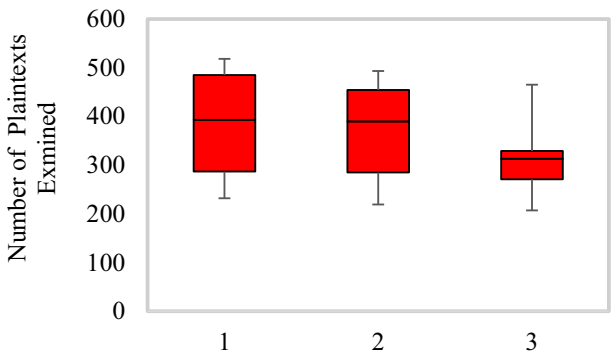

Fig. 3 Number of plaintexts examined for successful attacks (1: GA, 2: IGA, 3: NGA)

Table 3 Number of iterations required for successful attacks, fifty independent runs of GA, IGA and NGA algorithms. Data have been arranged in ascending order to indicate smallest (Min), Q1, Q2 (Median), Q3 and Largest (Max) values

GA	IGA	NGA	Remark	GA	IGA	NGA	Remark
7	7	4	Min	16	15	11	
7	7	4		17	15	11	
7	8	5		17	16	11	
8	8	5		18	16	12	
8	8	6		18	16	12	
8	8	7		18	17	12	
9	8	7		19	17	12	
9	9	8		19	17	13	
10	9	8		19	17	13	
10	9	8		19	18	13	
11	10	9		19	18	13	
11	10	9	Q1	20	18	14	Q3
12	10	9		20	18	14	
12	11	9		20	18	14	
13	11	9		20	19	14	
13	12	10		20	19	14	
14	12	10		21	19	15	
14	13	10		21	19	15	
15	13	10		21	20	15	
15	13	10		21	20	15	
15	14	10		22	20	15	
16	14	10		22	21	15	
16	14	10		22	21	15	
16	15	10		22	21	15	
16	15	11	Med.	23	21	15	Max

Table 4 Mean and standard deviation data corresponding to Table 3 for successful attacks

Method →	GA	IGA	NGA
Mean	15.72	14.48	10.92
Standard Deviation	4.78	4.36	3.13

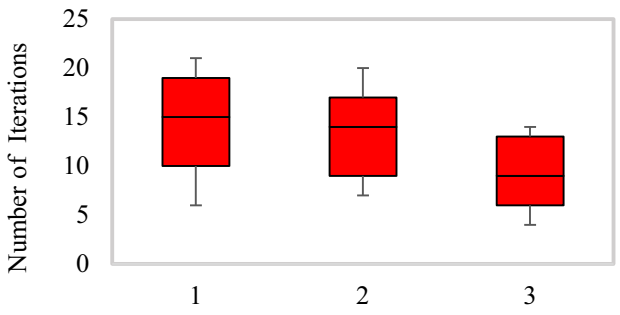

Fig. 4 Number of iterations required for successful attacks (1: GA, 2: IGA, 3: NGA)

Table 5 Statistical analysis of cryptanalytic results obtained via variants of the genetic algorithm

Method →	Comparison parameters	GA	IGA	NGA
Searched Space	Mean: μ_r	393.63	383.65	321.23
	S. Dev.: σ_r	99.15	97.12	61.17
Success Rate	Mean: μ_s	95.5	96.2	98.4
	S. Dev.: σ_s	1.13	0.81	0.79
Time	(in seconds)	62.65	59.23	12.38

respectively that are lesser comparatively to GA and IGA algorithms. From Table 4, it is observed that the average number of iterations required by NGA algorithm is 10.92, which is significantly lesser than 14.48 of IGA algorithm and 15.72 of GA algorithm. Moreover, the standard deviation is lower in the case of NGA algorithm. From Tables 3 and 4 and Fig. 4, we conclude that with respect to the criteria of the number of iterations required, NGA algorithm beats GA and IGA algorithms.

3. Average number of plaintexts examined (i.e., average searched space), average number of successful attacks and average performance time. For this criteria, all three algorithms have been tested 100 times on 100 random ciphertexts. Table 5 shows the statistics of cryptanalytic results, where the performance of each algorithm is determined for the following measures: (1) Mean (μ_r) and standard deviation (σ_r) with respect to the amount of state space searched. (2) Mean (μ_s) and standard deviation (σ_s) with respect to the number of successful attacks. (3) Average performance time in seconds. From Table 5, it is observed that the NGA algorithm outperforms GA and IGA algorithms. Results show that the success rate of NGA algorithm is better than GA and IGA algorithm. The success rate of NGA algorithm is 98.4%, while success rate of GA and IGA are 95.5 and 96.2%, respectively. The mean performance time taken by the NGA algorithm is 12.38 s which is significantly lesser than 62.65 s of GA and 59.23 s of IGA. It should be noted that though the mean amount of the searched space by all three algorithms is significantly better than the worst case brute force 1024, the statistical performance of the NGA algorithm for the amount of searched space is much better than GA and IGA algorithms ($\mu_r = 321.23$ and $\sigma_r = 61.17$ in the case of NGA, while $\mu_r = 383.65$ and $\sigma_r = 97.12$ in the case of GA and $\mu_r = 393.63$ and $\sigma_r = 99.15$ in the case of IGA).

Additionally, an experimental study is presented, where we analyze the performance of all three GA-based algorithms with respect to the fitness function used and the convergence speed (i.e., the number of iterations required and the time required to recover the exact plaintext). For

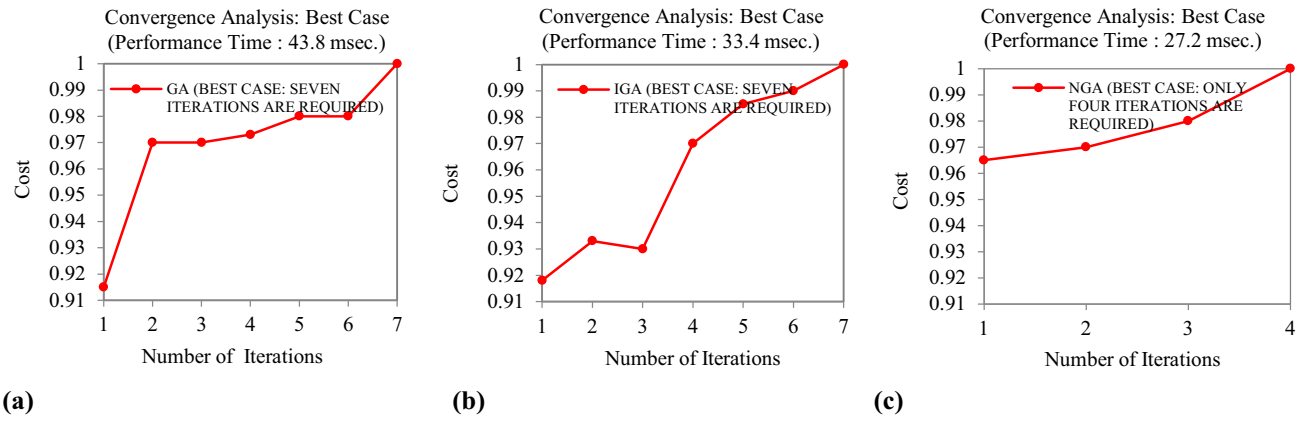

Fig. 5 Convergence analysis of genetic-based algorithms in the best case

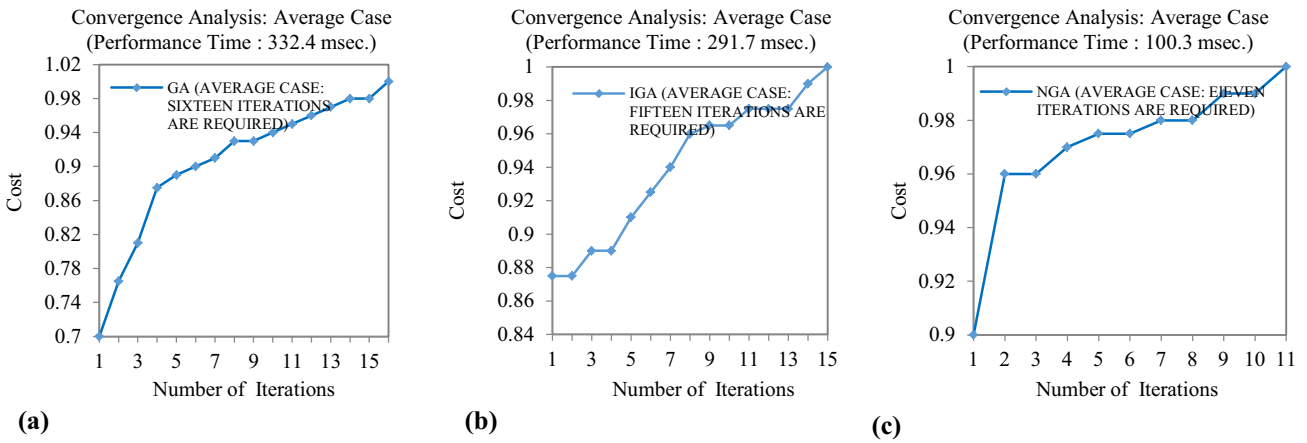

Fig. 6 Convergence analysis of genetic-based algorithms in the average case

this purpose e, a coin is flipped 10 times letting head (face = 1) and tail (face = 0). The plaintext = 0101100101 is obtained. This plaintext is then transformed into *Target* ciphertext using first 10 elements of the public key mentioned in Appendix B (the same method has been used to generate the random ciphertexts discussed in the above three criteria). The generated ciphertext have been attacked 50 times. The results in the best case, average case (considering median value) and worst case have been plotted in Figs. 5, 6 and 7, respectively. From these results, it is observe that in all the cases NGA algorithm beats GA and IGA algorithms. In the best case, 4 iterations are required by NGA algorithm, while GA and IGA algorithms require 7 iterations. In the average case, 11 iterations are required by NGA algorithm, while GA and IGA algorithms require 16 and 15 iterations, respectively. In the worst case, 15 iterations are required by NGA algorithm, while GA and IGA algorithms require 23 and 21 iterations, respectively. With respect to time: in the best case, 27.2 ms are required by NGA algorithm, while GA and IGA algorithms require 43.8 and 33.4 ms, respectively. In the average case, 100.3 ms are required by NGA algorithm, while GA and IGA algorithms require 332.4 and 291.7 ms, respectively. In the worst case, 105.5 ms are required by NGA algorithm, while GA and IGA algorithms require 398.9 and 374.1 ms, respectively. From this study, we conclude that the convergence of the NGA algorithm is significantly faster than GA and IGA algorithms The above-discussed results indicate that we should use the best GA, i.e., NGA algorithm along with NBCS and IGBPSO algorithms to demonstrate the attack on the practical RMKC scheme so that a fair comparison can be obtained between two classes, i.e., between evolutionary algorithm and swarm intelligence.

4.3 Cryptanalytic results obtained via NBCS, IGPBPSO and the best GA, i.e., NGA algorithms

For all the attack algorithms the limitation on population size and maximum number of iterations have been set as follows:

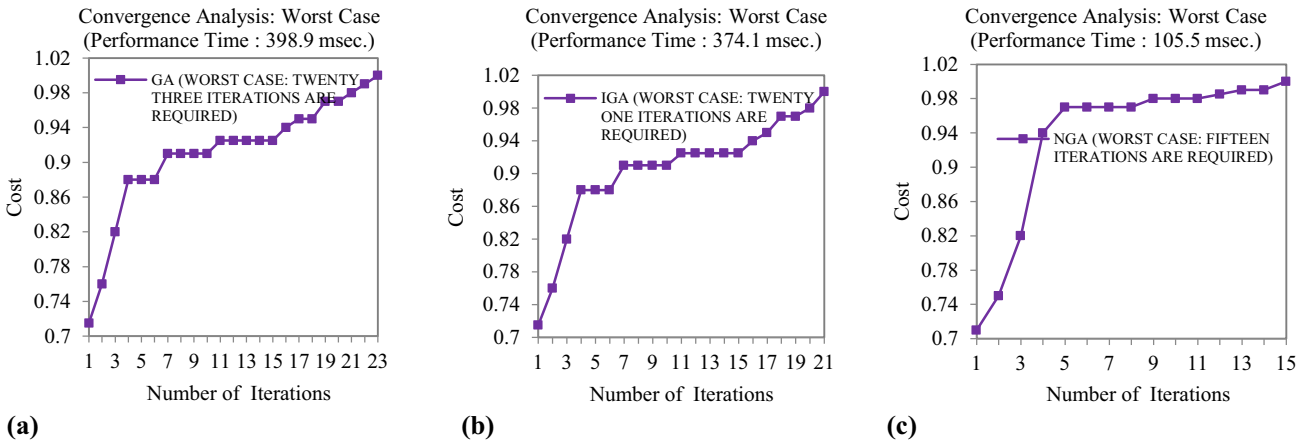

Fig. 7 Convergence analysis of genetic-based algorithms in the worst case

Table 6 Statistical analysis of cryptanalytic results of RMKC scheme obtained via NGA, IGBPSO and NBCS

Method ↓	Searched Space in %		Successful Attacks in %		Time in Minutes
	Mean	S. Dev.	Mean	S. Dev.	
NBCS	16.27	2.21	98.23	2.13	36.21
IGPBPSO	18.86	2.48	92.17	3.83	40.52
NGA	18.21	2.41	93.13	3.79	39.13

The maximum number of iterations is limited to 1000, and the limitation on the population size is set to the 1 Lac chromosomes or particles or host nests. The attacks have been mounted in a time frame manner using practical sized public key elements (as mentioned in Appendix B). That is, after a fixed number of iterations, either the exact plaintext will be determined or the algorithm will terminate with failure. From the perspective of a potential at tacker, the attack is significant, if it becomes successful in a given time limit. All the attack algorithms have been tested, 100 times on 100 different practical size ciphertexts. The statistics of cryptanalytic results obtained via each of these three algorithms are presented in Table 6. From Table 6, it is observed that the state space searched by all the algorithms is significantly lesser than the brute force attack 2^{40}. For achieving average success of 98.23%, the NBCS algorithm explore only 16.27% search space, while IGBPSO achieves 92.17% average success by exploring 18.86% search space, and NGA achieves 93.13% average success by exploring 18.21% search space. In terms of average time taken NBCS algorithm takes 36.21 min, while IGBPSO algorithm takes 40.52 min, and NGA algorithm takes 39.13 min. As evidence from the results, NBCS algorithm shows the superior performance than IGBPSO and NGA algorithms. These results indicate that NBCS algorithm is an effective and efficient choice for automated cryptanalysis of relative and similar cryptosystems.

5 Conclusion and future work

This paper has two objectives: first is to implement and possibly improve the previously studied genetic algorithm attacks of the knapsack cryptosystems. The second is to investigate the effectiveness and efficiency of the improved binary particle swarm optimization and cuckoo search techniques in automated cryptanalysis of the reduced complex knapsack cryptosystem. After several experiments, it is found that the adaptive fitness function along with the appropriate selection procedure, efficient crossover operator choice, and adaptive mutation operator significantly improves the performance of genetic algorithm. Through extensive cryptanalysis of the RMKC scheme, it is found that cuckoo search technique is the most effective and efficient choice. This study motivates that proposed algorithms, particularly NBCS offer a lot of promise in automated cryptanalysis of reduced cryptosystems. Therefore, these algorithms can be used as a first measure for automated cryptanalysis of the related cryptosystems. The presented algorithms can also be used for solving similar combinatorial problems, e.g., 0–1 knapsack problem, and similar security and privacy optimization problems. Subset cover problem is similar to subset sum problem and has a growing application in secure multi-party computation. An optimal covering of subset of countermeasures for privacy preservation during multi-party computation is an interesting and promising work that we plan to extend as a

future work by applying presented techniques. In addition, the proposed implementation model in Fig. 1 will help the researcher to design and analyze the public key cryptosystems that are based on IFP and DLP problems.

Appendix A

Algorithm 7: Determining a generator of Z_m^*

1. **Data:** a cyclic group Z_m^* of order p and the prime factorization $p = p_1^{e1} p_2^{e2} \ldots p_r^{er}$
2. **Result:** a generator g of Z_m^*
3. Randomly select an element of Z_m^* and assign it to g
4. **for** i from 1 to k **do**
5. Compute $temp \leftarrow g^{p/p_i} \bmod m$
6. **if** $temp = 1$ **then**
7. go to step 3. Comment: g is a generator of Z_m^* iff $g^{\emptyset(m/aFactor(m)} \equiv 1 \bmod m$
8. **end if**
9. **end for**
10. **return** g

Algorithm 8: Determining the order of an element in Z_m^*

1. **Data:** a group Z_m^* of order p, the prime factorization $p = p_1^{e1} p_2^{e2} \ldots p_r^{er}$, and an element $\alpha \in Z_m^*$
2. **Result:** the order β of α
3. **Set** $\beta \leftarrow p$
4. **for** i from 1 to k **do**
5. Compute $\beta \leftarrow \beta / {p_i}^{e_i}$
6. Compute $\alpha' \leftarrow \alpha^{\beta} \bmod m$
7. **while** $\alpha' \neq 1$ **do**
8. Compute $\alpha' \leftarrow (\alpha')^{p_i} \bmod m$
9. Compute $\beta \leftarrow \beta * p_i$
10. **end while**
11. **end for**
12. **return** β

Appendix B

See Table 7

Table 7 public Key Generated by RMKC Scheme

S.No.	Public Key elements Value (minimum 224-bit and maximum 227-bit representations)
1	1874556901448232271759650160508571049139639579325519679915728591999 97,
2	4811090193482739839348491950123625370243311390497196941273910468434 3,
3	4801280256522347959841065173079596162781917571038740381931295445973 0,
4	1234012543579013065192478522239569776280126388379855025394184321042 39,
5	7398693935492787699979042664401876852998183981331347926182387104356 9,
6	1777177182546853131283167921506271944884533224407514192738214928456 32,
7	2933676966507348964970420456606346345188700498240346915032841369103 8,
8	4549314965422612753265100793884755484904994639204701105346582173045 21,
9	7379929296371937149055184467703642106344124128385469239454610358345 76,
10	3909101996177018581712453715709733358196093773388512039532637728465 2,
11	3161643619387133485792971566308578793885281470774219215474376530753 4,
12	6862446501444751234103985641211890259505185020879015474968235494531 7,
13	6363178344683675330428804105845049247686294822831204624921677436576 72,
14	5549511478502238494936179145857656404219973920801202343434146467954 54,
15	1184493257621563520135811925043903984401367547636715340905293406322 5,
16	2885917083512935766164721122621351568595914956177144696480695064557 47,
17	6746464518380211765741173184019783250589107482786576786042465895566 3,
18	1610954845075095136604632519617867833480029635817108603502458686345 74,
19	8127688373070137757419715036309686833929168515800506750565675240569 64,
20	3851976431012334383071458567025836395403891437234057668052346507855 65,
21	1135908851717959134154802870914877330914620500911347684052487588056 3,
22	4729198273798999619427838343221414398463284760297175498679605046504 5,
23	1238475119125644468483739022302696935187392096850577476896592258695 7,
24	6519266383322787005628767151880052472034354476922028467956746067607 55,
25	5622364565917645755333872442352086479993291450877727978076606835356 32,
26	1493707354153242210516907011861060227771467699219606784565345138640 8,
27	5825435278872744974168332139766021553173700496752587986767467355476 7,
28	2020505073655100931147736953243930762531407690888341434572327895675 63,
29	1613452706219274975769862160493040279789530900946475657624486673524 52,
30	3536089455889208183577533364943520617960038855286778578563485231248 6,
31	1377179285691306126608189174143301270553731668454346884562434464842 32,
32	1606125806771225187102048809621675451641604916664101867675254367824 3,
33	1927861926221945334776935019012997493870112399588212358144986397882 57,
34	1841010310967080050652620062594760129733931776693559879269133357059 19,
35	1706189712918399543055468913574953807010757412514114309141464816588 57,
36	8179199362206398800264285622358142423482069834248034715211415423066 7,
37	1678862813501572228039444852970487422869870626397043681034735283843 21,
38	2054213783244196377168650119609564552375848994657675726363245335684 93,
39	4636199910873437152070673691806059003360927827027046874617658354365 7,
40	1765274393019785633766673686953738702691468687748370185398809870160 89

References

AbdulHalim MF, Bara'a AA, Hameed SM (2008) A binary particle swarm optimization for attacking knapsacks cipher algorithm. In: International conference on computer and communication engineering, pp 77–81, IEEE

Awad WS, El-Alfy ES (2015) Computational intelligence in cryptology. Improv Inf Secur Pract Through Comput Intell, 28–45

Bansal JC, Deep K (2012) A modified binary particle swarm optimization for knapsack problems. Appl Math Comput 218(22):11042–11061

Bhateja AK, Bhateja A, Chaudhury S, Saxena PK (2015) Cryptanalysis of vigenere cipher using cuckoo search. Appl Soft Comput 26:315–324

Boryczka U, Dworak K (2014) Genetic transformation techniques in cryptanalysis. In: Asian conference on intelligent information and database systems, Springer, pp 147–156

Danziger M, Henriques MA (2012) Computational intelligence applied on cryptology: a brief review. IEEE Lat Am Trans 10(3):1798–1810

Engelbrecht AP (2007) Computational intelligence: an introduction. Wiley, London

Forsyth WS, Safavi-Naini R (1993) Automated cryptanalysis of substitution ciphers. Cryptologia 17(4):407–418
Garg P, Shastri A (2006) An improved cryptanalytic attack on knapsack cipher using genetic algorithm. Int J Inf Technol 3(3):145–152
Goldberg DE, Holland JH (1988) Genetic algorithms and machine learning. Mach Learn 3(2):95–99
Gonzalez TF (2007) Handbook of approximation algorithms and metaheuristics. CRC Press, Boca Raton
Hei X, Song B (2014) SHipher: families of block ciphers based on subset-sum problem. IACR Cryptol ePrint Arch 2014:103
Jadon SS, Bansal JC, Tiwari R, Sharma H (2014) Artificial bee colony algorithm with global and local neighborhoods. Int J Syst Assur Eng Manag 1–13
Jain A, Bharadwaj A (2017) A genotype–phenotype binary particle swarm optimization technique with Lévy flights. In: ICONIP 2017, LNCS Springer (Accepted)
Jain A, Chaudhari NS (2014) Cryptanalytic results on knapsack cryptosystem using binary particle swarm optimization. In: International joint conference SOCO'14-CISIS'14-ICEUTE'14, Springer, pp 375–384
Jain A, Chaudhari NS (2015a) A new heuristic based on the cuckoo search for cryptanalysis of substitution ciphers. In: International conference on neural information processing, LNCS Springer, pp 206–215
Jain A, Chaudhari NS (2015b) Evolving highly nonlinear balanced boolean functions with improved resistance to dpa attacks. In: 9th International conference on network and system security, LNCS Springer, pp 316–330
Karagöz S, Yıldız AR (2017) A comparison of recent metaheuristic algorithms for crashworthiness optimisation of vehicle thin-walled tubes considering sheet metal forming effects. Int J Veh Des 73(1–3):179–188
Kate A, Goldberg I (2011) Generalizing cryptosystems based on the subset sum problem. Int J Inf Secur 10(3):189–199
Kennedy J, Eberhart RC et al. (1995) Particle swarm optimization. In: IEEE international conference on neural networks, vol 4, pp 1942–1948, IEEE
Kennedy J, Eberhart RC (1997) A discrete binary version of the particle swarm algorithm. In: IEEE international conference on systems, man, and cybernetics, vol 5, pp 4104–4108, IEEE
Khanesar MA, Teshnehlab M, Shoorehdeli MA (2007) A novel binary particle swarm optimization. In: International conference on control & automation, MED'07, pp 1–6, IEEE
Kiani M, Yildiz AR (2016) A comparative study of non-traditional methods for vehicle crashworthiness and NVH optimization. Arch Comput Methods Eng 23(4):723–734
Laskari EC, Meletiou GC, Stamatiou YC, Vrahatis MN (2007) Cryptography and cryptanalysis through computational intelligence. In: Computational intelligence in information assurance and security, Springer, pp 1–49
Lee S, Hong S (2016) Modified binary particle swarm optimization for multidimensional knapsack problem. Adv Sci Lett 22(11):3688–3691
Ma EY, Obimbo C (2011) An evolutionary computation attack on one-round TEA. Procedia Comput Sci 6:171–176
Mantegna RN (1994) Fast, accurate algorithm for numerical simulation of Levy stable stochastic processes. Phys Rev E 49(5):46–77
Martin KM (2017) Everyday cryptography: fundamental principles and applications. Oxford Press, Oxford
Matthews RA (1993) The use of genetic algorithms in cryptanalysis. Cryptologia 17(2):187–201
Menezes AJ, Van Oorschot PC, Vanstone SA (2010) Handbook of applied cryptography. CRC Press, Boca Raton
Merkle R, Hellman M (1978) Hiding information and signatures in trapdoor knapsacks. IEEE Trans Inf Theory 24(5):525–530
Michalewicz Z (2013) Genetic algorithms + data structures = evolution programs. Springer, New York
Muthuregunathan R, Venkataraman D, Rajasekaran P (2009) Cryptanalysis of knapsack cipher using parallel evolutionary computing. Int J Recent Trends Eng 1(1):3–6
Nalini N, Rao GR (2007) Attacks of simple block ciphers via efficient heuristics. Inf Sci 177(12):2553–2569
Palit S, Sinha SN, Molla MA, Khanra A, Kule M (2011) A cryptanalytic attack on the knapsack cryptosystem using binary firefly algorithm. In: 2nd International conference on computer and communication technology (ICCCT), pp 428–432, IEEE
Pampara G, Franken N, Engelbrecht AP (2005) Combining particle swarm optimisation with angle modulation to solve binary problems. In: IEEE congress on evolutionary computation, vol 1, pp 89–96, IEEE
Pholdee N, Bureerat S, Yıldız AR (2017) Hybrid real-code population-based incremental learning and differential evolution for many-objective optimisation of an automotive floor-frame. Int J Veh Des 73(1–3):20–53
Pohlig SC, Hellman ME (1978) An improved algorithm for computing logarithms over and its cryptographic significance. IEEE Trans Inf Theory 24(1):106–110
Sadri J, Suen CY (2006) A genetic binary particle swarm optimization model. In: IEEE congress on evolutionary computation, pp 656–663, IEEE
Sharma K, Chhamunya V, Gupta PC, Sharma H, Bansal JC (2015) Fitness based particle swarm optimization. Int J Syst Assur Eng Manag 6(3):319–329
Shi Y, Eberhart R (1998) A modified particle swarm optimizer. In: IEEE world congress on computational intelligence, pp 69–73, IEEE
Shor PW (1997) Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J Comput 41(2):303–332
Sinha SN, Palit S, Molla MA, Khanra A, Kule M (2011) A cryptanalytic attack on knapsack cipher using differential evolution algorithm. In: Recent advances in intelligent computational systems (RAICS), pp 317–320, IEEE
Spillman R (1993) Cryptanalysis of knapsack ciphers using genetic algorithms. Cryptologia 17(4):367–377
Spillman R, Janssen M, Nelson B, Kepner M (1993) Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers. Cryptologia 17(1):31–44
Srinivas M, Patnaik LM (1994) Genetic algorithms: a survey. Computer 27(6):17–26
Stinson DR (2005) Cryptography: theory and practice. CRC Press, Boca Raton
Vose MD (1999) The simple genetic algorithm: foundations and theory. MIT press, Cambridge
Wang B, Hu Y (2010) Quadratic compact knapsack public-key cryptosystem. Comput Math Appl 59(1):194–206
Wang B, Wu Q, Hu Y (2007) A knapsack-based probabilistic encryption scheme. Inf Sci 177(19):3981–3994
Yang XS (2014) Nature-inspired optimization algorithms. Elsevier, Amsterdam
Yang XS, Deb S (2009) Cuckoo search via Lévy flights. In: Nature and biologically inspired computing, NaBIC 2009, pp 210–214, IEEE
Yang XS, Deb S (2010) Engineering optimisation by cuckoo search. Int J Math Model Numer Optim 1(4):330–343
Yang XS, Cui Z, Xiao R, Gandomi AH, Karamanoglu M (2014) Swarm intelligence and bio-inspired computation: theory and applications, Newnes

Yildiz AR (2013a) Comparison of evolutionary-based optimization algorithms for structural design optimization. Eng Appl Artif Intell 26(1):327–333

Yildiz AR (2013b) Cuckoo search algorithm for the selection of optimal machining parameters in milling operations. Int J Adv Manuf Technol 64:55–61

Yildız AR (2009) Hybrid immune-simulated annealing algorithm for optimal design and manufacturing. Int J Mater Prod Technol 34(3):217–226

Yıldız AR (2009a) A novel hybrid immune algorithm for global optimization in design and manufacturing. Robot Comput Integr Manuf 25(2):261–270

Yıldız AR (2009b) An effective hybrid immune-hill climbing optimization approach for solving design and manufacturing optimization problems in industry. J Mater Process Technol 209(6):2773–2780

Yıldız BS (2017) A comparative investigation of eight recent population-based optimisation algorithms for mechanical and structural design problems. Int J Veh Des 73(1–3):208–218

Yıldız BS, Lekesiz H (2017) Fatigue-based structural optimisation of vehicle components. Int J Veh Des 73(1–3):54–56

Yildiz AR, Saitou K (2011) Topology synthesis of multicomponent structural assemblies in continuum domains. J Mech Des 133(1):1–9

Yildiz BS, Huseyin L, Ali RY (2016) Structural design of vehicle components using gravitational search and charged system search algorithms. Mater Test 58(1):79–81

Yıldız AR, Kurtuluş E, Demirci E, Yıldız BS, Karagöz S (2016) Optimization of thin-wall structures using hybrid gravitational search and Nelder–Mead algorithm. Mater Test 58(1):75–78