sciendo

BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **21**, No 2
Sofia • 2021 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2021-0019

A Recursive and Parallelized Dynamic Programming Implementation of Hard Merkle-Hellman Knapsack System for Public Key Cryptography

Vaddadi Sai Rahul, N. Narayanan Prasanth, S. P. Raja

School of Computer Science and Engineering, Vellore Institute of Technology, Vellore, Tamil Nadu, India
E-mails: nnprcd@gmail.com avemariaraja@gmail.com

***Abstract*:** *Merkle-Hellman public key cryptosystem is a long-age old algorithm used in cryptography. Despite being computationally fast, for very large input sizes it may operate slower due to thread creation overhead or reaching a deadlock situation. In this paper, we discuss the working principles of the Traditional Merkle-Hellman knapsack cryptosystem, which is an Easy knapsack. The challenges of Hard Knapsack and how it overcomes the shortcomings of the Traditional Easy Knapsack, are also discussed. The Hard knapsack variant of Merkle-Hellman is solved first using plain recursion and then improvised using a dynamic programming approach to the problem. Parallelism and Concurrency has been achieved on the dynamic programming implementation using OpenMP API which further has enhanced the performance time. A comparative study of both variants of Hard Knapsack for messages of different lengths has shown that the latter is faster.*

***Keywords*:** *Merkle-Hellman public key cryptosystem, Easy knapsack, Hard knapsack, recursion, dynamic programming, parallelism, OpenMP API.*

1. Introduction

The Knapsack problem is an example of a combinatorial optimization problem, which seeks for a best solution from among many other solutions. It is concerned with a knapsack that has a positive integer volume (or capacity) *W*. There are *N* distinct items that may potentially be placed in the knapsack, each of which has an assigned value. The Merkle-Hellman system utilizes a similar concept of Knapsack. The Merkle-Hellman knapsack public key cryptosystem, invented in 1970's by Whitfield Diffie, Martin Hellman and Ralph Merkle, was one of the earliest public key cryptosystems. During its proposal in 1970's it emerged as one of the fastest and secure encryption systems. Although Merkle-Hellman is not secure compared to

other widely used algorithms like RSA, DSA, SHA-512, etc., it is really fast. In 1984, Adi Shamir's cryptanalysis of Merkle-Hellman has shown that the system can be breached easily and hence, has been discarded ever since. The vulnerabilities in the system can be attributed to the constraints imposed.

The Merkle-Hellman system is bounded mainly by two constraints. First, the set of knapsack elements must be super increasing – current element's value is greater than the sum of its previous elements in the sequence. This is called an *easy knapsack* sequence in Merkle-Hellman. On the contrary, a *hard knapsack* sequence is one in which elements do not follow the super-increasing constraint. Unlike any other public key cryptosystem, Merkle-Hellman system has two keys – a public key for encryption and a private key for decryption. The public key is a hard knapsack and the private key is an easy knapsack. Due to this constraint, the problem is easily solvable in polynomial time. Second, the capacity of the knapsack is fixed; and must be greater than sum of all the knapsack elements. The system utilizes two numbers – a multiplier and a modulus, in the processes of public key generation and message decryption.

Another variant of the Merkle-Hellman cryptosystem involves using a hard knapsack even for public key. As mentioned earlier, in a hard knapsack, the set of input knapsack elements are free from the super-increasing constraint, i.e., random selection of elements is supported. This randomized approach accounts for the receiver of the encrypted message solving an instance of the *subset sum* problem during decryption phase, which is NP-Hard. The major drawback to this approach however, is the exponential nature of recursion.

For real-life applications, the messages to be sent are practically very large in size, making the entire process of encryption and decryption inefficient for a recursive solution. An alternate approach for converting a plain recursion to an iterative version is by Dynamic Programming (DP). This optimization algorithm helps in shrinking the exponential time complexity of recursion to polynomial time. The dynamic programming implementation can be further catalysed by using several parallelization techniques. OpenMP is a lightweight and scalable model, which provides a simple and versatile interface to parallel programmers for developing portable parallel applications. It supports parallel multi-platform shared-memory programming on all architectures including Unix and Windows Platforms in C, C^{++} and Fortran. We can add vectorized implementations by using the pragma directives present in OpenMP, which help in catalysing the decryption process; hence making it very efficient [1]. This paper presents an exact recursive and parallelized dynamic programming implementation for solving the Hard knapsack. The proposed method overcomes the constraint limitations by using a randomized public key and weight selection.

The working of traditional Merkle-Hellman easy knapsack is provided in Section 3 of this paper. The proposed Recursive and Parallel Dynamic Programming implementations of Hard Knapsack are illustrated in Section 5 of this paper. Finally, in the last Section 6 a comparison between the performance of Recursive and Parallel dynamic programming implementations of Merkle-Hellman hard knapsack for varying message lengths is presented as an illustration.

2. Related work

The Merkle-Hellman knapsack cryptosystem is mostly confronted with security issues, but much less in regards to the execution time. Many attacks were successful in breaching the system. As this can be impractical for real world applications, there have been many proposed approaches, which have been formulated to bolster the security breaches in system, and also in reducing the time complexity of Knapsack problem. A few of these are listed below.

To mask the easy knapsack series, Yuan Ping et al. [2] use a Chinese remainder theorem. The implicit intruder must therefore solve at least two challenging number theoretical problems in order to recover the information of the trapdoor, namely Integer factorization and Diophantine approximation problems. Patcharin Buayen and Jeeraporn Werapun [3] suggests a parallel time–space reduction of polynomial time to solve the 0-1 knapsack. This focuses on achieving quick processing by unbiased-filtering as the FS (Feature Sorting) and the optimal solutions are similar to the DP where $O(n/p\log_2 p + C'/p)$ is the optimized time-complexity.

An improved version of Shamir's attack on the basic Merkle-Hellman cryptosystem based on orthogonal lattice technique has been proposed by Liu, Bi and Xu [4]. This new concept would help in estimating the security of new cryptosystems based on Knapsack. An exact algorithm for the knapsack problem in cryptography called as The Packing Tree Search algorithm has been provided by Slonkina, Kupriyashin and Borzunov [5] and this proved to be one of the best algorithms in Knapsack cryptography. The algorithm seems to be scalable, although efficiency of parallel computations can be as small as 50-60%.

It uses the dynamic packing weight estimate, which takes into account faster step-based tree traversal techniques with the substitution of linearization-based and stack-based tree traversal techniques. To solve large scale 0-1 knapsack problems, Zhou and Zhao [6] have presented the Social Spider Optimization (SSO) Algorithm. The SSO algorithm is based on the simulation of social spiders' cooperative behaviour. In this algorithm, individuals emulate a group of spiders who communicate in accordance with the cooperative colony's biological laws.

By introducing a simple quadratic compact Knapsack problem, Baocang wan and Yupu hu [7] have proposed a knapsack-type public key cryptosystem. This method uses the Chinese remainder theorem to mask the sequence of the simple knapsack. The system's encryption function is nonlinear with respect to message vector. The system enjoys high density under there-linearization attack model. A discrete logarithm-based public-key cryptosystem has been proposed by Paar and Pelzl [8]. The striking feature is that the computational time and size of cipher text levelled that of the RSA scheme, whereas it equalled the Elgamal Cryptosystem in terms of security level.

Nguyen and Stern [9] notice that the frequent presence of low density underlying in knapsacks, made them vulnerable to lattice attacks, both in theory and practice. By introducing non-binary coefficients in addition to reducing the weight of knapsack, they avoid low-density attacks by increasing the density of the knapsack

beyond some critical density. A probabilistic Quantum Poly-Time (QPT) Turing Machines modelling of the sender and receiver by employing classical channels between them has been proposed by Okamoto, Tanaka and Uchiyama [10] – a new paradigm known as Quantum public-keycryptosystem. It uses a quantum trapdoor-function that allows a QPT machine to calculate fwith high probability.

Cai and Cusick [11] have presented a Lattice-Based Public key cryptosystem that recently identified a random class of integer point lattices that could prove the following worst-case average-case equivalence result:

"If there is a probabilistic polynomial time algorithm which finds a short vector in a random lattice from the class, then there is also a probabilistic polynomial time algorithm which solves several problems related to the Shortest lattice Vector Problem (SVP) in any *n*-dimensional lattice".

3. Merkle-Hellman public key cryptosystem architecture

We need two different keys in Merkle-Hellman's public key knapsack cryptosystem. One key is Public while the other is Private. The Public key is used in the process of Encryption, and helps to encrypt (or encode) the message. Since it is "public", anyone can use it. The Private Key is used in the process of Decryption and helps in decoding (or to decrypt) the encrypted message. This key is kept secret (or private) so that the message can only be decrypted by the person who knows the key. The person with the Private Key may also encrypt a message with the Private Key, and then someone with the Public Key may decrypt that message.

The system has three main modules:

1. Public and Private Key generation.
2. Encryption process.
3. Decryption process.

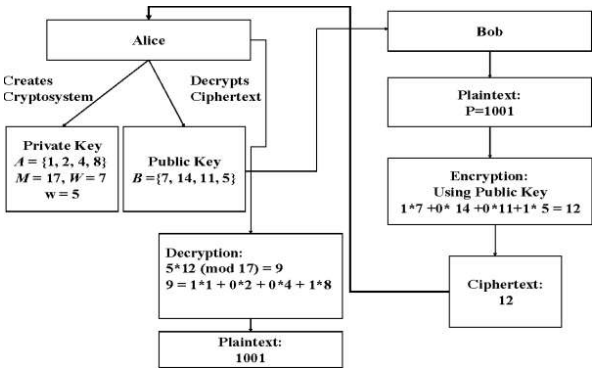

Fig. 1. Merkle-Hellman knapsack cryptosystem architecture

Public and private Key generation: Private key for the system is the knapsack elements itself, which is a super-increasing sequence. If *S* is a knapsack containing *n* elements $w_1, w_2, w_3, \ldots, w_n$ and *P* is a private key, then

$$S = (w_1, w_2, w_3, \ldots, w_n),\ P = S = (w_1, w_2, w_3, \ldots, w_n).$$

A super-increasing sequence is a collection of values, in which current value must be greater than the sum of all its previous values in the same collection. Let w_j be an arbitrary element of index j in knapsack S. The super-increasing sequence satisfies the condition

$$\sum_{i=1}^{j-1} w_i < w_j, \qquad 1 < j \leq n.$$

Public key is generated by taking the knapsack sequence and multiplying all the values by a number, W is called *the multiplier*, and modulo m is called *the modulus*. The modulus should be a number greater than the sum of all the numbers in the sequence. The multiplier and the modulus must be co-prime. This is called an Easy knapsack sequence. Let B be the generated public key, then

$$B = (b_1, b_2, b_3, \ldots, b_n) \mid b_i = (w_i W) \mathrm{mod} m, 1 \leq i \leq n;$$
$$m > \sum_{i=1}^{n} w_i, \ \gcd(W, m) = 1.$$

The pseudocode for generating private and public key is given below in Listing 1 and Listing 2, respectively.

Listing 1. generate_private_key function
Function 1: generate_private_key
Input: *arr
Output: returns private key
return arr

Listing 2. generate_public_key function
Function 2: generate_public_key
Input: *arr, *public_key, *N*, *m*, *W*
Output: returns public key
For i = 0 to *N* do
public_array[i] = (arr[i]**W*) mod *m*
end for
return public_array

Encryption. Encryption is carried out with the help of public key. Message to be sent is encrypted by splitting the message into groups of *n*, *n* being the number of knapsack elements. Let *M* be a message of *k* bits $m_1, m_2, m_3, \ldots, m_k$ with m_1 being the highest order bit. Let message be divided into groups $g_1, g_2, \ldots, g_{\frac{k}{n}}$,

$$g_i = \left(m_{1+n(i-1)}, \ldots, m_{ni}\right), \quad 1 \leq i \leq k/n\ .$$

The position of bits corresponding to 1 in each group g_i are looked up in the public key B to find the combination of elements and added up. This generates the encoded message or ciphertext. Let c be the generated ciphertext formed by combining ciphers c_i of each group g_i,

$$c = \{c_i\};\ c_i = \sum_{\mathrm{j}=1}^{n} b_{\mathrm{j}}\, g_{ij}, \quad 1 \leq i \leq \frac{k}{n}.$$

The pseudocode for the Encryption process is given below in Listing 3.

Listing 3. Encryption function
Function 3: Encryption
Input: *N*, *message, public_key, encrypted, mssg_length
Output: returns encrypted message
Declare sum, *i*, *j*, pos=0
For *i* = 0 to mssg_length/*N* do
Initialize sum to 0
Initialize *j* to *i***N*
While *j*=0 to (*i*+1) **N* and message[*j*] not equal to null do
If message[*j*] is equal to 1 then
Increment sum by public_key[*j*%*N*]
End if
Increment *j* by 1
End while
Initialize encrypted[pos] to sum
Increment pos by 1
end For
return encrypted

Decryption. The first stage of Decryption process involves generating W^{-1}, where W^{-1} is multiplicative inverse of W. The receiver should know both the numbers W and m. The inverse is obtained by solving the equation

(1) $$W(W^{-1}) = 1 \mod m.$$

The calculated multiplicative inverse is then multiplied by each element of the ciphertext c_i, generating some numbers. Let c' be the decrypted message

$$c' = \{c'_i\} \mid c'_i = c_i W^{-1} \mod m, \qquad 1 \le i \le k/n.$$

As this is an Easy knapsack system, i.e., elements are in super-increasing sequence, we can easily find the subset of elements in private key which equal the obtained numbers in c' in polynomial time ($O(n)$) by using *Extended Euclidian Algorithm* [13]. Let X = $(x_1, x_2, x_3, \dots, x_t)$ be the list of indices of elements in private key *P* which sum to c'_i, i.e.,

$$c'_i = \sum_{j=1}^{t} P_{x_j} \mid c'_i \in c'.$$

The message *M* is then obtained by setting $(P_{x_1}, P_{x_2}, P_{x_3}, \dots, P_{x_t}) \to 1$ and rest to 0.

$$M = \{P_{x_i}\} \mid P_{x_i} = 1,\ x_i \in X;\quad P_{x_i} = 0,\ x_i \mid \in X;\ 1 \le i \le n.$$

Example. Consider $S = \{1, 2, 4, 10, 20, 40\}$, $W = 31$ and $m = 110$,

$b_1 =$ (1×31) mod 110 = 31,
$b_2 =$ (2×31) mod 110 = 62,
$b_3 =$ (4×31) mod 110 = 14,
$b_4 =$ (10×31) mod 110 = 90,
$b_5 =$ (20×31) mod 110 = 17,
$b_6 =$ (40×31) mod 110 = 30,

$$B = (31, 62, 14, 90, 17, 30); \quad P = S = (1, 2, 4, 10, 20, 40).$$

Let $M =$ '100100111100101110', k = 18 and n = 6 => k/n = 3,

$$g_1 = 100100,$$
$$g_2 = 111100,$$
$$g_3 = 101110,$$
$$c_1 = 31 + 90 = 121,$$
$$c_2 = 31 + 62 + 14 + 90 = 197,$$
$$c_3 = 31 + 14 + 90 + 70 = 205,$$
$$c = (121, 197, 205).$$

By solving Equation (1), we get $W^{-1} = 71$,

$$c'_1 = 121 \times 71 \bmod 110 = 11 \rightarrow X = (x_1, x_4) = 100100,$$
$$c'_2 = 197 \times 71 \bmod 110 = 17 \rightarrow X = (x_1, x_2, x_3, x_4) = 111100,$$
$$c'_3 = 205 \times 71 \bmod 110 = 35 \rightarrow X = (x_1, x_3, x_4, x_5) = 101110,$$
$$\therefore M = 100100\ 111100\ 101110.$$

4. Hard knapsack and its challenges involved over the Traditional Merkle-Hellman knapsack cryptosystem

The Traditional Merkle-Hellman knapsack cryptosystem involves something known as an Easy knapsack. An Easy knapsack is a sequence in which each element in the sequence is greater in value than sum of all its previous element values. This knapsack is solvable in *O*(*n*) polynomial time using a greedy algorithm known as Extended Euclidean Algorithm. In a super-increasing sequence, there always exists a unique subset for a particular target sum in the decryption process. So, determining the combination of elements can be done in a single pass. Another requirement of this system involves selecting a weight element for the knapsack, which is fixed and must be greater than sum of all the knapsack elements. Therefore, Merkle-Hellman System involves constraints of generating a super-increasing sequence and choosing a specific weight.

There exists another variant of the Merkle-Hellman knapsack cryptosystem, which uses a Hard knapsack. The Hard knapsack involves generating a random weight and a random knapsack sequence, contrary to the super-increasing constraint of the traditional Easy knapsack. As the sequence of knapsack elements are randomly generated, determining the combination of elements which tantamount to the target sum involves solving an instance of the Subset Sum problem, which in practice is known to be NP-Hard.

5. Methodology

The approach taken towards the problem in the paper is divided into two parts:

– the first part involves solving the Hard Knapsack of the Merkle-Hellman Public Key Cryptosystem in exponential time using a recursive implementation of an algorithm called Subset sum;

– and the second part involves optimizing the exponential nature of the Subset sum using dynamic programming and further parallelizing the optimized solution using OpenMP's pragma distribute Single Instruction Multiple Data (SIMD) construct.

This proposed system has first and second modules similar to the Traditional Merkle-Hellman knapsack cryptosystem – Public and Private key generation and Encryption process as given in Section 3. The third module deals with decryption process of a Hard knapsack followed by optimization and parallelization. In the proposed algorithm, selection of knapsack elements and weight of the knapsack are at the sender's discretion as opposed to the super-increasing sequence and a fixed weight in the Easy knapsack of the Traditional Merkle-Hellman public key cryptosystem. Let $w_i \epsilon R_I$, where R_I represents a set of random integers. For a Hard knapsack,

$$S = \{w_i \mid w_i \epsilon R_I\},$$
$$W = r \mid r \epsilon R_I.$$

5.1. Data recursive solution of Hard knapsack in Merkle-Hellman cryptosystem

Subset Sum problem is an important decision problem in cryptography [12]. The problem involves finding a subset of elements whose totality equals the given target sum. In Merkle-Hellman cryptosystem, private key represents the subset and the sequence of numbers generated during the decryption process, individually represents target sum. The general solution to this problem is recursive in nature. The pseudocode for the Decryption process in Hard knapsack and recursive subset sum implementation is given below in Listings 4, 5 and 6.

Listing 4. Decryption function
Function 4: Decryption function
Input: *encrypt_pointer, *W*, *m*, encrypt_items, *private_key, *N*
Output: Call to findSubset(set [], size, sum)
Initialize *n*_inverse = call to find_multiplicative_inverse(*W*, *m*) function
Initialize *i* = 0
Declare decrypt
While *i* = 0 to encrypt_items do
decrypt=(encrypt_pointer[*i*]**n*_inverse) mod *m*
call to findSubset(private_key,*n*,decrypt)
increment *i* by 1
Initialize *j* to *i***N*
End While

Listing 5. findSubset function
Function 5: findSubset function
Input: set [], size, sum
Output: call to subsetSum(set, subset, size, 0, 0, 0, sum)
Declare a subSet pointer and dynamically allocate memory equal to size
Call to subsetSum(set, subset, size, 0, 0, 0, sum)
Free(subset)

Listing 6. subsetSum (Recursion) function
Function 6: subsetSum function
Input: set [], subSet [], N, subSize, total, nodeCount, sum
Output: generate decrypted message
Declare *j*
If total is equal to sum then
Call to displaySubSet (subset, subSize, set, *N*)
Return
Else
For *j* = nodeCount to *N* do
subset[subSize] = set[*j*]
Recursive Call to subsetSum(set, subset, *N*, subSize+1, total+set[*j*], *j*+1, sum)
End for
End if

5.2. Application of dynamic programming and parallelization techniques for reduced time complexity of Hard knapsack

The recursive solution for the Subset Sum problem of Hard knapsack works fine for smaller message lengths. In real world applications, the message to be sent from the sender to the receiver side is very large. This turns out to be a rather time-consuming process. So, dynamic programming algorithm is an optimization to the recursive solution of subset sum. It reduces the overhead of exponential time complexity to polynomial time. Further, concurrency and parallelism are achieved using OpenMP's pragma distribute simd construct. The parallel construct used is: *# pragma omp distribute simd*. The omp distribute simd directive achieves concurrent execution of loop iterations, by distributing these iterations to each master thread which adheres to the SIMD (Single Instruction Multiple Data) instructions. Therefore, each master thread is assigned different elements of knapsack, wherein the elements of each thread are summed up and compared with the target sum in a concurrent fashion. The pseudocode for the PDP (Parallelized Dynamic Programming) implementation of subset sum is given below in Listing 7.

Listing 7. subsetSum (dynamic programming +parallelization) function
Function 7: findSubset
Input: arr [], *N*, *k*
Output: generate decrypted message
Declare *i*, *j*, elem_pos[*N*]
For *i*=0 to *N* do
Initialize elem_pos[*i*]=0
End for
Initialize *m* to pow (2, *n*)
#pragma omp distribute simd
For *i*=*m* – 1 till greater than 0 do
Initialize sum=0 and *b*=*i*
For *j*=0 to *N* do

Increment sum by (b mod 2) *arr[j]
Divide b by 2 and assign it to b
End for
If sum is equal to k
Initialize b to i
For j=0 to N do
If (b mod 2) then assign elem_pos[j] to 1
Divide b by 2 and assign it to b
End for
End if
End for
Initialize original_mssg to 0
For j=0 to N do
Update original_mssg = pow (2, j) *elem_pos[N – 1 – j] + original_mssg
Display original_mssg

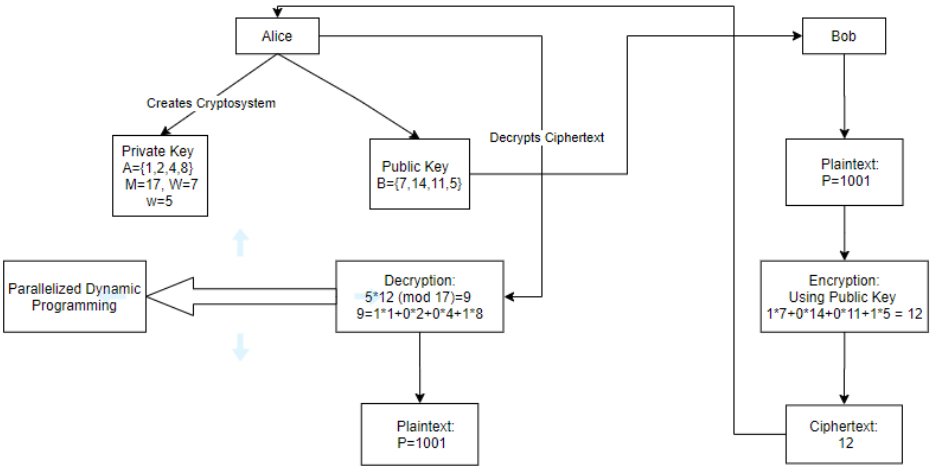

Fig. 2. Application of dynamic programming and parallelization to Public key knapsack cryptosystem

6. Results and discussions

Both the Recursive implementation and Parallel Dynamic Programming (PDP) implementation of the Hard knapsack in Merkle-Hellman Public Key cryptosystem have been tested on an Intel i3-7100U CPU 7th Gen Processor with a Clock Speed @2.40GHz. It is a dual-core processor with two threads per core. The input message is a string of characters. Hence, each character is converted to their ASCII format and represented in bits. Since ASCII value of a character in English Language is of 8-bits, the number of elements in knapsack is kept constant – 8. Random knapsack elements are taken as {180, 7, 2, 21, 11, 354, 89, 42} and having a random capacity of 706.

Table 1. Performance time of Recursive and Parallel dynamic programming implementations of Hard knapsack for varying message length

Length of message (letter count)	Average time taken (recursive Hard knapsack)	Average time taken (PDP) Hard knapsack
50	0.004 s	0.004 s
100	0.009 s	0.008 s
150	0.013 s	0.012 s
200	0.021 s	0.014 s
250	0.019 s	0.018 s
500	0.04 s	0.031 s
1000	0.079 s	0.070 s

The performance of both implementations for seven varying message lengths is shown above in Table 1. Apparently, for small message lengths, we observe that the execution time for both the algorithms is almost similar. However, as the message length increases, the latter runs approximately ~1.15 times faster than the former. A graphical representation of the performance time for both algorithms in shown below in Fig. 3.

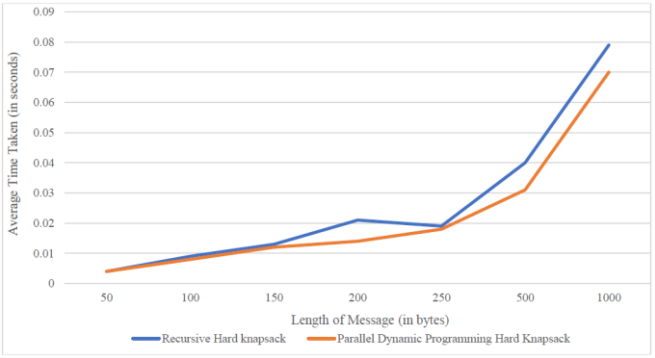

Fig. 3. Comparative performance of Parallel and Serial Merkle-Hellman Knapsack public key cryptosystem for different sizes of message

7. Conclusion

The paper discusses the performance of Merkle-Hellman Easy knapsack system and various challenges involved in the Hard knapsack of Merkle-Hellman. A recursive and parallelized dynamic programming versions of Hard knapsack variant of Merkle-Hellman have been implemented. Parallelism has been obtained using OpenMP pragma constructs. Simulation results show that the parallelized dynamic programming implementation has obtained a larger degree of concurrency and hence achieves maximum performance in terms of time. Comparative analysis of both variants of Hard knapsack for different message sizes gives an insight that the PDP implementation is faster. The novelty of this work relates to the properties of Hard Merkle-Hellman knapsack – hard public key sequence and dynamic weight selection. This might require an attacker solving two instances – a hard sequence for determining the public key; and the capacity of the knapsack, leading to an increased security for the proposed system. Since the objective of this paper is focused on

speeding up the Hard knapsack of Merkle-Hellman system, the security of the proposed algorithm forms basis for future research.

References

1. M a l i k, M., S. M a l h o t r a, N. P r a s a n t h. Time Improvement of Smith-Waterman Algorithm Using OpenMP and SIMD. – FTNCT 2019, CCIS 1206, 2020, pp. 686-697. **https://doi.org/10.1007/978-981-15-4451-4_54**
2. P i n g, Y., B. W a n g, S. T i a n, J. Z h o u, H. M a. Towards aProbabilistic Knapsack Public-Key. – Cryptosystem with High Density, Information, Vol. **10**, 2019, No 2, 75. **https://doi.org/10.3390/info10020075**
3. B u a y e n, P., J. W e r a p u n. Parallel Time–Space Reduction by Unbiasedfiltering for Solving the 0/1-Knapsack Problem. – Journal of Parallel and Distributed Computing, Vol. **122**, December 2018, pp. 195-208. **https://doi.org/10.1016/j.jpdc.2018.08.003**
4. L i u, J., J. B i, S. X u. An Improved Attack on the Basic Merkle-Hellman KnapsackCryptosystem. – IEEE Access, Vol. **7**, 2019, pp. 59388-59393. DOI:10.1109/ACCESS.2019.2913678.
5. S l o n k i n a, I., M. K u p r i y a s h i n, G. B o r z u n o v. Analysis and Optimization of the PackingTree Search Algorithm for the Knapsack Problem. – In: Proc. of IEEE Conference of Russian YoungResearchers in Electrical and Electronic Engineering (EIConRus'19), Saint Petersburg and Moscow, Russia, 2019, pp. 1811-1815. DOI: 10.1109/EIConRus.2019.8657309.
6. Z h o u, Y. Q., R. Z h a o. Solving Large-Scale 0-1 Knapsack Problem by the Socialspideroptimisation Algorithm. – January International Journal of Computing Science and Mathematics, Vol. **9**, 2018, No 5, pp. 433-441. DOI: 10.1504/IJCSM.2018.095497.
7. B a o c a n g, w a n, Y u p u h u. Quadratic Compact Knapsack Public-Key Cryptosystem. – Computers & Mathematics with Applications, Vol. **59**, 2019, No 1, pp. 194-206. **https://doi.org/10.1016/j.camwa.2009.08.031**
8. P a a r, C., J. P e l z l. Public-Key Cryptosystems Based on the Discrete Logarithm Problem. – In: Understanding Cryptography. Berlin, Heidelberg, Springer, 2010. **https://doi.org/10.1007/978-3-642-04101-3_8**
9. N g u y ễ n, P. Q., J. S t e r n. Adapting Density Attacks to Low-Weight Knapsacks. – In: B. Roy, Ed. Advances in Cryptology – ASIACRYPT 2005. ASIACRYPT 2005. Lecture Notes in Computer Science. Vol. **3788**. Berlin, Heidelberg, Springer, 2005. **https://doi.org/10.1007/11593447_3**
10. O k a m o t o, T., K. T a n a k a, S. U c h i y a m a. Quantum Public-Key Cryptosystems. – In: Proc. of 20th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '00). Berlin, Heidelberg, Springer-Verlag, 2000, pp. 147-165.
11. C a i, J. Y., T. W. C u s i c k. A Lattice- Based Public-Key Cryptosystem. – In: S. Tavares, H. Meijer, Eds. Selected Areas in Cryptography. SAC 1998. Lecture Notes in ComputerScience. Vol. **1556**. Berlin, Heidelberg, Springer, 1999. **https://doi.org/10.1007/3-540-48892-8_18**
12. R a g h u n a n d a n, K. R., A. G a n e s h, S. S u r e n d r a, K. B h a v y a. Key Generation Using Generalized Pell's Equation in Public Key Cryptography Based on the Prime Fake Modulus Principle to Image Encryption and Its Security Analysis. – Cybernetics and Information Technologies, Vol. **20**, 2020, No 3.
13. **https://www.wikiwand.com/en/Merkle%E2%80%93Hellman_knapsack_cryptosystem**

Received: 29.12.2020; Second Version: 16.03.2021; Accepted: 30.03.2021