Parallelized Common Factor Attack on RSA

Vineet Kumar[1]*, Aneek Roy[1], Sourya Sengupta[1], Sourav Sen Gupta[2]

[1] Jadavpur University, Kolkata
[2] Indian Statistical Institute, Kolkata

The actual version is at https://link.springer.com/chapter/10.1007/978-3-319-72598-7_18

Abstract

In this paper, we present a parallel approach to *common factor attack* on RSA moduli obtained by mining TLS and SSH certificates from the Internet. Our work generalizes that of Heninger et al. (2012) for a resource constrained environment, where the memory may not be sufficient to create the *product tree* required for batch-wise GCD computation on the entire dataset. We propose a data-parallel routine to efficiently exploit the batch-wise GCD algorithm in a resource constrained setting, and mount the common factor attack on TLS and SSH certificates to obtain the set of vulnerable RSA moduli with reasonable accuracy.

1 Introduction

RSA [1], invented in 1978 by Rivest, Shamir and Adleman, is the most widely used public key cryptographic primitive to date. RSA is used as a trapdoor permutation to build encryption schemes as well as digital signatures. The strength of RSA depends upon the intractability of factoring its modulus N, which is a product of two randomly chosen *large* primes p and q. According to SSL Pulse [2], around 93% of the sites surveyed in the recent past uses a 2048-bit RSA modulus, or equivalent. Although PKCS#1 standard for RSA does not mandate primes of the same bit-size, the primes are generally chosen to be of equal bit-size in practical implementations. Thus, the sites surveyed by SSL Pulse most commonly use 1024-bit primes p and q.

The best known classical algorithm for factoring the RSA modulus is the General Number Field Sieve (GNFS), with a computational complexity for an integer n as follows [3]:

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}}+o(1)\right)(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}\right)=L_n\left[\frac{1}{3},\sqrt[3]{\frac{64}{9}}\right].$$

This means that the computational strength of a 2048-bit RSA is approximately 116 bits (112 bits as per NIST) and that of a 1024-bit RSA is approximately 86 bits (80 bits as per NIST). This claimed strength of RSA (or factoring) against GNFS is valid if the primes composing the RSA modulus are chosen uniformly at random from a full entropy distribution. For example, in case of a 1024-bit RSA, it is expected that the primes p and q are chosen independently and uniformly at random from all possible 512-bit primes, satisfying $p \neq q$.

Note that according to the Prime Number Theorem, there are of the order of 2^{502} primes of bit-size 512. Thus, if two primes p and q are chosen independently and uniformly at random from all possible 512-bit primes, then the chance of getting the same prime twice (with probability 0.5) is approximately 2^{-256} (birthday collision probability). This intuitively obvious idea is challenged by the recent *common factor attacks* on RSA. The reader may refer to [4, 5, 6] in this direction.

1.1 Common factor attack on RSA

In 2012, Heninger *et al.* [4] and Lenstra *et al.* [5] introduced the idea of this attack. They performed an Internet wide survey, mined all TLS and SSH certificates, and performed an exhaustive pairwise-GCD computation including every RSA modulus thus obtained. It is intuitively expected that two 1024-bit RSA moduli $N_1 = p_1q_1$ and $N_2 = p_2q_2$ will be co-prime with probability close to 1, and $\gcd(N_1, N_2)$ will reveal one of the primes with probability close to 2^{-256}, as is the case with prime collision. However, Heninger *et al.* [4] found an overwhelming number of primes revealed through the mass GCD computation. They obtained the primes and private keys for about 0.50% of TLS hosts and about 0.03% of SSH hosts, as their RSA moduli shared common primes with that of the other hosts. This was an alarming demonstration of an unnatural vulnerability in RSA usage across the Internet, primarily caused by low-entropy randomness extraction.

*vntkumar8@gmail.com

Heninger *et al.* [4] traced the cause of this vulnerability to *sloppy* implementations of RSA in embedded systems, especially in routers, firewalls, and other network devices. In case of random prime generation, RSA implementations tend to use pseudo-random number generators (PRNGs) seeded by fresh randomness each time. However, as the smaller network devices try to generate the primes at boot, quite often they lack a full-entropy source for extracting the random seeds, and hence the primes they generate eventually have a much higher probability of collision. This low-entropy phenomenon is more pronounced in case of devices manufactured by the same vendor, or in devices produced in the same batch, as the prime generation routine in such devices operate identically.

In practice, Heninger *et al.* [4] found that 0.75% of TLS certificates shared RSA primes, and conjectured that another 1.70% were susceptible to similar compromise due to sloppy implementations. Lenstra *et al.* [5] reported similar results too. Later in 2013, following the footsteps of [4, 5], Bernstein *et al.* [6] demonstrated that the RSA public keys embedded in the smart cards of Taiwan's national "Citizen Digital Certificate" database can also be compromised due to similar vulnerabilities. Bernstein *et al.* [6] used batch-wise GCD computation, batch-wise trial division, as well as more sophisticated tools such as Coppersmith-type partial-key-recovery attacks, to compromise a significant number of these RSA moduli.

1.2 Motivation of our work

The recent trend in computing is to move from traditional single-core processing units to massive multi-core computing architectures, primarily targeted at more processing power harnessed through efficient parallelization and pipelining. This trend has seen the rise of GPUs, multi-core CPUs, and hybrid architectures around the globe. In case of *data parallel computation*, the process is parallelized by distributing the data between the compute nodes, in contrast with *task parallel computation*, where the distribution of the computation task is emphasized over the distribution of data. Single instruction, multiple data (SIMD) framework is ideal for data parallel computation, where multiple processors perform same instruction on different pieces of data, as in a distributed file system.

Heninger *et al.* [4] performed the pairwise GCD computation of the scanned RSA moduli using the best known algorithm for such computations – *batch-wise GCD* [7]. It performs the required pairwise GCD computation by building product and remainder trees (as shown in Figure 1) with expected computational complexity of $O(n(\log n)^2 \log(\log n))$, where n is the number of RSA moduli in the dataset. Thus, for a large dataset comprising of more than 4 to 5 Million RSA moduli, it requires almost 32 GB of memory and around 60 to 70 GB of storage for scratch calculations [4]. This is the worst computational bottleneck of the approach followed in [4].

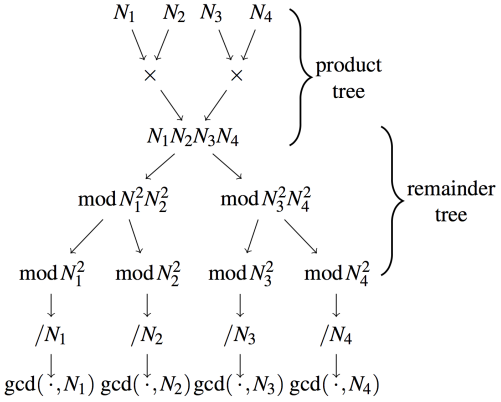

Figure 1: Batch GCD computation technique used in [4] (figure from [4]).

In 2016, Heninger *et al.* [8] proposed a partially parallel implementation of the batch-wise GCD algorithm, where the dataset is partitioned into subsets and the product tree is constructed individually for each subset. While this provides full parallelization in the first phase of the algorithm, the remainder tree is still constructed considering all subset products (as shown in Figure 2), which acts as a computational bottleneck.

In contrast, we choose a data parallelization approach for both the product tree and the remainder tree computations, thereby parallelizing the complete batch-wise GCD algorithm.

1.3 Contribution of our work

In this paper, we propose an alternative approach to achieve similar results as Heninger *et al.* [4, 8], using less computational resources. With the progress from serial to parallel computing frameworks, we choose to adopt

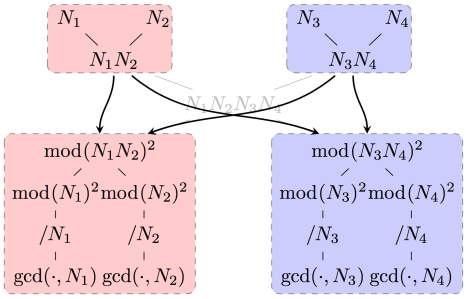

Figure 2: Parallel GCD computation technique used in [8] (figure from [8]).

the *data parallelization* approach in order to reduce the memory and storage complexity of the batch-wise GCD algorithm in practice.

Consider the master dataset D comprising of several RSA moduli scanned from the Internet. Suppose that the size of D is large enough so that in a regular resource constrained node, one cannot run batch-wise GCD computations on D. Suppose that a single node can handle a dataset of size at most m, where $m \ll |D|$. The solution that we provide splits the dataset D into $p \sim |D|/m$ partitions, so that each node may operate on the partitions in parallel, and then takes a union of results obtained from several such *random* splits of the data to identify the final set of vulnerable RSA moduli. We prove a bound on the fraction of vulnerable moduli identified by this approach, and show that one may probabilistically extract almost all vulnerable RSA moduli quite efficiently in practice.

The paper is organized as follows. Section 2 summarizes the proposed *parallelized common factor attack* — including the modified batch-wise GCD algorithm (Algorithm 1), the theoretical proof for the accuracy of our algorithm (Theorem 1), and the experimental results (Section 2.2) to support our claim. Finally, Section 3 concludes the paper, illustrating a few scopes and directions for future investigation.

2 Parallelized Common Factor Attack

The proposed attack has three modules – partitioning of data, computation on the subsets, and aggregation of results. In the first phase, the dataset D is partitioned into $p \sim |D|/m$ partitions, where m is the largest data-size that a single node can handle. This m remains a parameter for the user to choose.

The computation of the product and remainder trees to identify vulnerable RSA moduli is performed individually on each subset, using the free and open source code (available at `http://factorable.net`) for batch-wise GCD. Note that this method does not recover all vulnerable moduli present in D, as there may be several pairs of moduli with common factor split into different subsets. Thus, we require a mechanism to aggregate the results obtained from each subset of the data.

We take the union of the sets of vulnerable moduli obtained from each subset of the data D, thus eliminating duplicates, if any. To recover all vulnerable moduli in the master dataset D, we repeat this process over a number of iterations k.

Later in this section, we prove a probabilistic lower bound on the fraction of recovered moduli as a function of the number of partitions (p) and the number of iterations (k). This helps us to choose the optimal number of iterations (k) in our attack, based on the number of partitions (p), size of each partition (m), and the desired level of accuracy (ϵ). The desired level of accuracy (ϵ) acts as another user-defined parameter.

Algorithm 1 presents the proposed parallelized common factor attack, and Figure 3 illustrates one complete iteration of our proposed algorithm, distributed over the three modules.

2.1 Accuracy of the algorithm

We claim that for a *proper* choice of k, using the subroutine `chooseIteration` in Algorithm 1 recovers approximately ϵX vulnerable RSA moduli, if there are a total of X vulnerable moduli in the input dataset D, and ϵ is the pre-specified user-defined accuracy parameter. The *proper* choice of k is proposed and justified in Theorem 1.

Theorem 1. *Suppose that there are X vulnerable RSA moduli in an input dataset D. Then Algorithm 1 recovers an expected number of ϵX vulnerable RSA moduli if we choose*

$$k \approx \frac{\log(1-\epsilon)}{\log m + \log(p-1) - \log(mp-1)}$$

Input : Set of moduli D, constraint m, accuracy ϵ
Output: V — set of vulnerable moduli in D

$p \leftarrow$ `ceiling`$(|D|/m)$;
$k \leftarrow$ `chooseIteration`$(m,\ p,\ \epsilon)$;
for $i \leftarrow 1$ **to** k **do**
  $\{d_1, d_2, \ldots, d_p\} \leftarrow$ `randomPartition`$(D,\ p)$;
  $\{v_1, v_2, \ldots, v_p\} \leftarrow$ `batchGCD`$(\{d_1, d_2, \ldots, d_p\})$;
  $V_i \leftarrow$ `setUnion`$(\{v_1, v_2, \ldots, v_p\})$;
end
$V \leftarrow$ `setUnion`$(\{V_1, V_2, \ldots, V_k\})$;

Algorithm 1: Parallelized Common Factor Attack

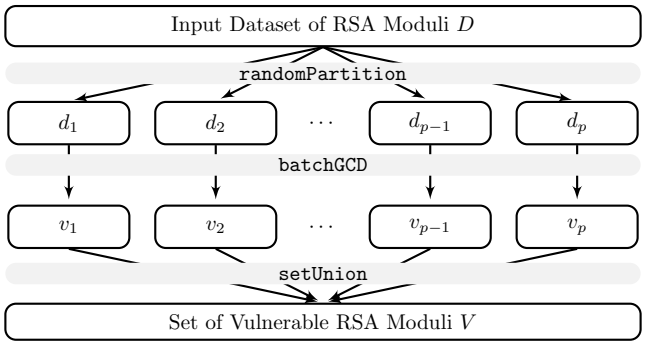

Figure 3: One complete iteration of the proposed Parallelized Algorithm.

where ϵ is the user-defined accuracy parameter, m is the user-defined constraint, and $p \sim |D|/m$ is the number of partitions.

Proof. Let us define an undirected graph G_D on input dataset D, with the RSA moduli N_i as vertices, and edges present between vertices N_i, N_j if and only if $\gcd(N_i, N_j) > 1$. Thus, finding the set of vulnerable RSA moduli in D amounts to finding all the edges of the graph G_D, and this is precisely the output of the `batchGCD` algorithm executed on D.

Note that p random partitions $\{d_1, d_2, \ldots, d_p\}$ of the input dataset D partitions the graph G_D into mutually exclusive subgraphs $\{g_1, g_2, \ldots, g_p\}$ (as illustrated in Fig. 4), each with approximately $m \sim |D|/p$ vertices. If we execute `batchGCD` in parallel on each subset in $\{d_1, d_2, \ldots, d_p\}$, we will obtain all edges within each subgraph in $\{g_1, g_2, \ldots, g_p\}$ (illustrated as solid edges in Fig. 4), but will miss the edges $e_{(N_i,N_j)}$ in G_D where N_i and N_j belong to two different subgraphs in $\{g_1, g_2, \ldots, g_p\}$ (illustrated as dotted edges in Fig. 4).

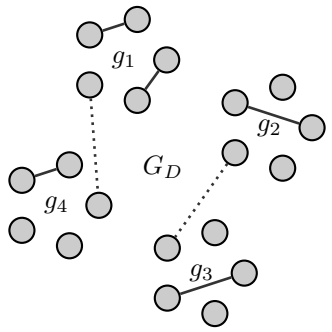

Figure 4: Illustrative partition of graph G_D into subgraphs $\{g_1, g_2, \ldots, g_p\}$.

Given that the `randomPartition` subroutine randomly partitions the input dataset D into p (approximately) equal subsets $\{d_1, d_2, \ldots, d_p\}$, the probability that we will miss a specific edge $e_{(N_i,N_j)}$ in G_D after one execution

of the parallel `batchGCD` algorithm on $\{d_1, d_2, \ldots, d_p\}$ can be computed as

$$\begin{aligned} P_{i=1} &= 1 - \frac{\text{total number of edges in } \{g_1, g_2, \ldots, g_p\}}{\text{total number of edges in } G_D} \\ &\approx 1 - \frac{\text{edges in complete supergraph of } \{g_1, g_2, \ldots, g_p\}}{\text{edges in complete supergraph of } G_D} \\ &\approx 1 - \frac{p \times \binom{m}{2}}{\binom{mp}{2}} = 1 - \frac{m-1}{mp-1} = \frac{m(p-1)}{mp-1}, \end{aligned}$$

where the edges in the graph G_D are assumed to exist uniformly at random, depending on the existence of common factors between the moduli (vertices). Therefore, the probability that we will miss a specific edge $e_{(N_i,N_j)}$ in G_D after k independent executions of the parallel `batchGCD` algorithm on k independent random partitions $\{d_1, d_2, \ldots, d_p\}$ of D is

$$P_{i=k} = (P_{i=1})^k \approx \left(\frac{m(p-1)}{mp-1}\right)^k,$$

and thus, the fraction of edges recovered after k iterations is

$$\epsilon \approx 1 - \left(\frac{m(p-1)}{mp-1}\right)^k.$$

Hence the choice for k prescribed in the theorem. □

Note that given the dataset D and that $mp \sim |D|$, one may also interpret the values of ϵ and k from Theorem 1 as

$$\begin{aligned} \epsilon &\approx 1 - \left(\frac{|D| - |D|/p}{|D| - 1}\right)^k, \quad \text{that is,} \\ k &\approx \frac{\log(1-\epsilon)}{\log(|D| - |D|/p) - \log(|D| - 1)} \end{aligned}$$

so that the choice of k in the subroutine `chooseIteration` depends directly on the input dataset. However, the two interpretations of ϵ and k are identical, and thus one may write the subroutine `chooseIteration` in either way. One may note the relationship between k, the desired level of accuracy ϵ, and the number of partitions p, from the following experimental results.

2.2 Experimental results

We downloaded the set of TLS certificates of secured websites from the Internet-wide scan data repository available at `https://www.scans.io` [9]. The dataset contained around 27 million 1024-bit RSA moduli ($\sim$ 7 GB in size) and around 15 million 2048-bit RSA moduli ($\sim$ 6 GB in size).

To perform a proof-of-concept validation of our proposed parallelized algorithm, we chose a random 500 MB set of 1024-bit RSA moduli as our input dataset D. The choice of 500 MB dataset D was based on our constraints of computational resources, as the machine used for the experiments (Intel Core i5 4210U CPU, 4 GB RAM) could only process 500 MB of RSA moduli at a time, using naive `batchGCD` [4].

To check the accuracy of our proposed algorithm, we executed the naive `batchGCD` algorithm on the 500 MB dataset D to know the exact number of vulnerable RSA moduli X, and then executed the proposed algorithm on D with various values of p (2 to 32, in powers of 2) and k (1 to 9) to obtain the experimental values of ϵX. Running this experiment multiple times on our 500 MB random dataset D provided us with a distribution of ϵ, given the parameters p and k.

The experimental distribution of ϵ, given parameters p and k, is recorded in Table 1, and the expected (mean) relationship of ϵ with the parameters p and k is illustrated in Figure 5.

The experimental results clearly indicate that the proposed parallelized common factor attack is capable of recovering more than 90% of the vulnerable RSA moduli from a given dataset D even when the dataset has been partitioned into $p = 16$ parts, provided that we allow multiple iterations; for example, $k = 8$ iterations in case of $p = 16$. For a fixed number of partitions p, there is a clear monotonic relationship between ϵ and k, indicating the possibility of recovering *almost all* vulnerable RSA moduli from a given dataset, by executing our proposed algorithm for a suitable number of iterations.

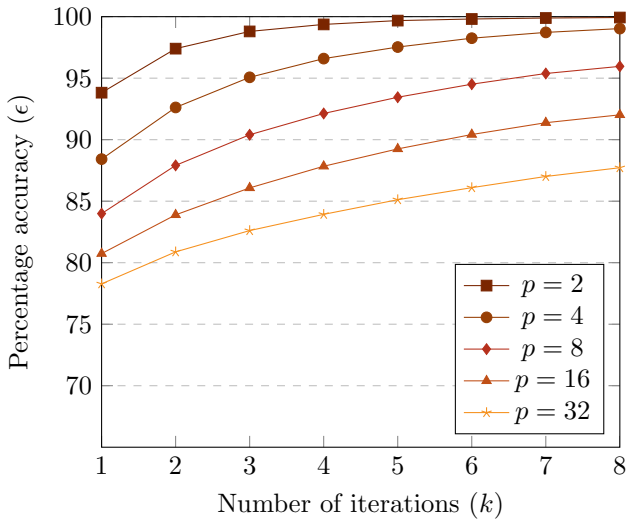

Figure 5: Relationship between ϵ, p and k from experimental data.

3 Conclusion

In this paper, we generalize the `batchGCD` algorithm of Heninger *et al.* [4] towards a parallelized common factor attack on RSA moduli. Our proposal is naturally scalable over arbitrary sized datasets of a similar nature, and one may consider implementing the same over inherently parallel data processing frameworks like GPU clusters or Hadoop map-reduce platforms. As a follow up of our work, one may consider extending our proposal to include the partially parallel approach of Heninger *et al.* [8] as well. Another interesting line of future work may be to extend our proposal to include the sophisticated approaches of finding vulnerable RSA moduli (using Coppersmith type attacks), in lines with the work of Bernstein *et al.* [6].

References

[1] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[2] SSL Labs, "Trustworthy Internet movement – SSL pulse," 2017, online; last accessed 17 April 2017. [Online]. Available: https://www.trustworthyinternet.org/ssl-pulse/

[3] C. Pomerance, "A tale of two sieves," *Notices Amer. Math. Soc*, vol. 43, pp. 1473–1485, 1996.

[4] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your ps and qs: Detection of widespread weak keys in network devices," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, T. Kohno, Ed. USENIX Association, 2012, pp. 205–220. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger

[5] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, "Ron was wrong, whit is right," *IACR Cryptology ePrint Archive*, vol. 2012, p. 64, 2012. [Online]. Available: http://eprint.iacr.org/2012/064

[6] D. J. Bernstein, Y. Chang, C. Cheng, L. Chou, N. Heninger, T. Lange, and N. van Someren, "Factoring RSA keys from certified smart cards: Coppersmith in the wild," in *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, ser. Lecture Notes in Computer Science, K. Sako and P. Sarkar, Eds., vol. 8270. Springer, 2013, pp. 341–360. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-42045-0_18

[7] D. J. Bernstein, "How to find smooth parts of integers," 2004. [Online]. Available: http://cr.yp.to/papers.html#smoothparts

[8] M. Hastings, J. Fried, and N. Heninger, "Weak keys remain widespread in network devices," in *Proceedings of the 2016 ACM on Internet Measurement Conference, IMC 2016, Santa Monica, CA, USA, November*

14-16, 2016, P. Gill, J. S. Heidemann, J. W. Byers, and R. Govindan, Eds. ACM, 2016, pp. 49–63. [Online]. Available: http://dl.acm.org/citation.cfm?id=2987486

[9] Censys Team, University of Michigan, "Internet-wide scan data repository," 2016, online; last accessed 15 October 2016. [Online]. Available: https://scans.io/

Table 1: Experimental results for ϵ given the parameters p and k.

k					$p = 2$					**mean**	std.dev
1	93.87	93.87	93.71	93.83	93.91	93.72	93.75	93.85	93.85	93.82	0.07
2	97.54	97.35	97.45	97.45	97.30	97.34	97.54	97.43	97.17	97.40	0.12
3	98.79	98.80	98.88	98.73	98.74	98.90	98.77	98.81	98.75	98.80	0.06
4	99.40	99.41	99.35	99.28	99.46	99.39	99.41	99.35	99.31	99.37	0.06
5	99.72	99.64	99.62	99.70	99.69	99.67	99.61	99.76	99.73	99.68	0.05
6	99.81	99.81	99.83	99.86	99.76	99.81	99.83	99.79	99.80	99.81	0.03
7	99.88	99.89	99.92	99.87	99.89	99.87	99.89	99.93	99.90	99.89	0.02
8	99.93	99.94	99.92	99.96	99.93	99.96	99.94	99.90	99.93	99.93	0.02

k					$p = 4$					**mean**	std.dev
1	88.19	88.44	88.51	88.56	88.34	88.32	88.74	88.33	88.32	88.42	0.16
2	92.77	92.61	92.77	92.45	92.60	92.72	92.50	92.50	92.66	92.62	0.12
3	95.13	95.30	94.94	95.09	95.16	95.00	95.04	95.12	94.87	95.07	0.13
4	96.69	96.48	96.61	96.65	96.37	96.49	96.55	96.66	96.78	96.59	0.13
5	97.51	97.68	97.69	97.47	97.57	97.60	97.58	97.26	97.41	97.53	0.14
6	98.30	98.31	98.04	98.17	98.38	98.22	98.29	98.19	98.33	98.25	0.10
7	98.85	98.66	98.75	98.62	98.79	98.73	98.83	98.64	98.63	98.72	0.09
8	99.05	99.18	98.97	98.99	99.08	99.14	99.07	98.74	99.05	99.03	0.13

k					$p = 8$					**mean**	std.dev
1	84.05	84.05	84.23	83.66	83.85	84.08	84.13	84.01	83.82	83.99	0.18
2	87.95	87.93	87.77	87.90	87.97	88.06	88.01	87.89	87.71	87.91	0.11
3	90.28	90.35	90.48	90.47	90.51	90.51	90.37	90.27	90.36	90.40	0.10
4	92.01	92.09	92.10	92.17	92.13	92.12	92.19	92.17	92.17	92.13	0.06
5	93.37	93.40	93.45	93.47	93.42	93.45	93.47	93.49	93.51	93.45	0.05
6	94.46	94.45	94.51	94.49	94.54	94.56	94.49	94.57	94.52	94.51	0.04
7	95.31	95.38	95.33	95.38	95.32	95.43	95.37	95.48	95.44	95.38	0.06
8	96.07	95.99	96.10	96.11	96.17	96.10	95.87	95.21	96.07	95.96	0.29

k					$p = 16$					**mean**	std.dev
1	80.69	80.62	81.08	80.75	80.55	80.89	80.60	80.72	80.74	80.74	0.16
2	83.77	84.25	83.89	83.72	84.13	83.68	83.82	83.92	83.82	83.89	0.19
3	86.26	85.96	85.87	86.20	85.80	85.95	86.06	86.38	86.29	86.08	0.20
4	87.94	87.81	88.13	87.79	87.95	88.04	87.55	87.88	87.50	87.84	0.21
5	89.21	89.48	89.14	89.30	89.35	89.25	89.06	89.19	89.25	89.25	0.12
6	90.43	90.20	90.35	90.39	90.46	90.58	90.69	90.32	90.34	90.42	0.15
7	91.26	91.40	91.45	91.21	91.23	91.44	91.50	91.61	91.31	91.38	0.14
8	92.14	92.17	92.07	92.19	92.05	92.25	92.03	91.17	92.14	92.02	0.33

k					$p = 32$					**mean**	std.dev
1	78.54	78.28	78.32	78.01	78.29	78.22	78.33	78.08	78.46	78.28	0.17
2	80.96	80.88	80.67	80.94	80.82	80.91	80.76	81.07	80.94	80.88	0.12
3	82.57	82.44	82.73	82.59	82.69	82.55	82.82	82.45	82.65	82.61	0.13
4	83.84	84.02	83.97	84.02	83.86	84.09	83.95	83.80	83.83	83.93	0.10
5	85.10	85.03	85.05	85.00	85.13	85.20	85.21	85.10	85.26	85.12	0.09
6	86.13	86.13	86.09	86.18	86.02	85.99	86.13	86.04	86.15	86.10	0.06
7	86.98	86.96	87.07	87.01	87.08	86.89	86.99	87.01	87.06	87.01	0.06
8	87.75	87.83	87.91	87.91	87.87	87.83	87.76	86.85	87.75	87.72	0.33