Journal of Cryptographic Engineering manuscript No.
(will be inserted by the editor)

A Preliminary FPGA Implementation and Analysis of Phatak's Quotient-First Scaling Algorithm in the Reduced-Precision Residue Number System

Christopher D. Nguyen · Dhananjay S. Phatak · Steven D. Houston · Alan T. Sherman

Received: date / Accepted: date

Abstract We built and tested the first hardware implementation of Phatak's Quotient-First Scaling (QFS) algorithm in the reduced-precision residue number system (RP-RNS). This algorithm is designed to expedite division in the Residue Number System for the special case when the divisor is known ahead of time (i.e., when the divisor can be considered to be a constant, as in the modular exponentiation required for the RSA encryption/decryption). We implemented the QFS algorithm using an FPGA and tested it for operand lengths up to 1024 bits. The RP-RNS modular exponentiation algorithm is not based on Montgomery's method, but on quotient estimation derived from the straightforward division algorithm, with substantial amount of precomputations whose results are read from look-up tables at run-time.

Phatak's preliminary analysis indicates that under reasonable assumptions about hardware capabilities, a single modular multiplication's (or QFS's) execution time grows logarithmically with respect to the operand word length. We experimentally confirmed this predicted growth rate of the delay of a modular multiplication with our FPGA implementation. Though our implementation did not outperform the most recent implementations such as that by Gandino, et al., we determined that this outcome was solely a consequence of tradeoffs stemming from our decision to store the lookup tables on the FPGA.

Our work provides useful design information for future hardware implementations and we interpret our results as promising for the RP-RNS algorithms.

Keywords Reduced-Precision Residue Number System · Residue Number System (RNS) · modular exponentiation · Quotient-First Scaling (QFS) algorithm · computer arithmetic · FPGA hardware

C.D. Nguyen
E-mail: cn1@umbc.edu

D.S. Phatak
E-mail: phatak@umbc.edu

S.D. Houston
E-mail: stevenh2@umbc.edu

A.T. Sherman
Cyber Defense Lab
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
E-mail: sherman@umbc.edu

1 Introduction

Modular exponentiation is a key operation in many of today's public-key cryptography and digital signature algorithms. Recently, Phatak has introduced a new modular exponentiation algorithm based on a reduced-precision residue number system (RP-RNS) [10–13]. RP-RNS is attractive for its inherent parallelism and time-space tradeoff achieved by augmenting computation with pre-stored approximations in intermediate calculations. To quickly perform the modular multiplications of the RP-RNS modular exponentiation algorithm, Phatak developed a novel method for dividing by a constant in RP-RNS called Quotient-First Scaling (QFS) [11]. Using an FPGA, we implemented Phatak's RP-RNS QFS algorithm and experimentally confirmed the logarithmic growth of its execution time with respect to the operand word length. For additional background and explanation about our work, see Nguyen [8].

Residue number systems (RNSs) represent integers as residues with respect to a set of pair-wise coprime integers called moduli. In the RNS, Addition/Subtraction and multiplication are parallel operations, i.e., they can be performed in each residue-channel independent all other channels. On the other hand, magnitude comparison or a sign-detection and consequently a division are slower and more complex in RNSs than in the binary number system. The promising speed of RNSs, together with this curious Area(Lookup-

Table 1 Algorithm Notation

$\hat{x}$	approximation of x
m_i	ith modulus
K	number of moduli
$\mathbf{M}$	$\{m_1, m_2, \ldots, m_K\}$; the RNS base
M	$\prod_{i=1}^{K} m_i$
$\hat{m}$	$\sum_{i=1}^{K} m_i/K$
N	$\lg M$; word size of the operands
ρ_0, Rc_Z	reconstruction coefficient of Z
$\overline{Z}$	Z in RP-RNS form
Z_i, $[Z]_{m_i}$	ith residue of Z
M_i	M/m_i
M_i^{-1}	inverse of M_i modulo m_i
ρ_i	$Z_i M_i^{-1} \bmod m_i$
f_i	ρ_i/m_i

tables) vs. speed tradeoff (which can yield further speedup beyond currently known methods), makes modular exponentiation in RNSs an interesting subject of research.

Currently, there are two approaches to RNS-based modular exponentiation. Most approaches are based on Montgomery's method [7], and have focused on improving the speed of the base extension operation [1–4,6,9,5] since that is the limiting step in the algorithm when deployed in RNSs.

Alternatively, Phatak's RP-RNS approach uses reduced precision to perform a fast scaling [11] without resorting to base extensions or Montgomery's method (as long as the RP-RNS moduli are chosen properly). As detailed in Section 3, Phatak's RP-RNS approach requires fewer residue operations but uses additional precomputed storage.

We experimentally verify Phatak's preliminary analysis of the RP-RNS Quotient-First Scaling algorithm, claimed to run in $\mathrm{O}(\lg N)$ cycles, where N is the word length of the operands. Since QFS is performed in each of the $\mathrm{O}(N)$ iterations of the RP-RNS modular exponentiation algorithm, evidence of the QFS algorithm's run-time will give credence to the claimed $\mathrm{O}(N \lg N)$ run-time for the modular exponentiation algorithm [12].

We adopt notation consistent with Phatak (see Table 1). For clarity, we will sometimes use ρ_0 in lieu of Rc_Z. We refer to the RNS moduli set as the RNS base.

An RP-RNS uses the RNS base consisting of K consecutive prime numbers starting at either 2 or 3 together with an additional 1- or 2-bit redundant modulus respectively, denoted m_e. The RP-RNS has integer range $[0, M-1]$. A number Z in RP-RNS form consists of its residues over $\mathbf{M}$ and m_e. The RNS as defined motivates our need for custom hardware since the number of moduli needed for cryptographic purposes exceeds the parallelism offered by conventional hardware. For example, the RP-RNS modular exponentiation algorithm uses 234 moduli for 1024-bit divisors and over 1500 moduli for 4096-bit divisors.

Our contributions include experimental data and analyses, which support Phatak's preliminary analyses; and the lessons learned; which should benefit future hardware implementations of the RP-RNS algorithms. Implementing Phatak's algorithms in hardware is non-trivial because the RP-RNS base includes moduli of varying non-standard word lengths (i.e., the smallest primes).

2 Algorithms

Assuming a fixed divisor D, instead of performing Montgomery multiplication, the QFS algorithm performs regular modular multiplication and directly uses the division algorithm (Equation 1) to perform the reductions modulo D:

$$Z = QD + R, \tag{1}$$

where Q and R are the quotient and remainder of Z divided by D respectively. The quotient-first scaling (QFS) algorithm (stated in full as "Algorithm 1" in the Appendix below; for details see [11,13]) computes a quotient estimate $\hat{Q} \in \{Q-1, Q\}$. This calculation yields a remainder estimate $\hat{R} \in \{R, R+D\}$ respectively, which is congruent to Z mod D. The QFS algorithm depends on the partial reconstruction algorithm [10, 13].

Phatak makes the following assumptions, which we satisfy:

- choose M such that $9D^2 < M$;
- beginning with 2 or 3, use K consecutive prime numbers for $\mathbf{M}$; and
- choose $m_e = 2$ if $2 \notin \mathbf{M}$, and $m_e = 4$ otherwise.

The reason for the first assumption is that if $9D^2 < M$ then we can chain modular multiplications using $\hat{R}$ without performing magnitude comparison and correction until the very end of a modular exponentiation.

The QFS algorithm lookup tables require $\mathrm{O}\left(N^3 \lg\lg N\right)$ bits of storage where N is the word length of the operands. Under Phatak's hardware assumptions, the QFS algorithm should run in $\mathrm{O}(\lg N)$ time. We have replicated the algorithm for reference in Appendix A; see Phatak [10–13] for proofs of correctness and detailed analyses.

3 Performance Comparison

An analytical performance comparison of Phatak's modular exponentiation algorithm to other RNS modular exponentiation algorithms is shown in Tables 2, 3, and 4. For each of Phatak's [11], Kawamura et al.'s [6], and Bajard et al.'s [4,5] modular exponentiation algorithms, we compare the number of residue multiplications, residue additions, small-bit scalar additions, and pre-computation storage required. The operation count given is the total operation count and not the critical path length (with additional hardware, some of the operations can be performed in parallel). The notation

Table 2 Performance of Phatak's ME algorithm [11] (without correction)

Operation	Total Count
Residue multiplications	$\Theta(NK)$
Residue additions	$\Theta(NK^2)$
Small-bit additions	$\Theta(NK)$
Precomputation storage (bits)	$\Theta(NK\hat{m})$

Table 3 Performance of Kawamura et al.'s ME algorithm [6] (without correction)

Operation	Total Count
Residue multiplications	$\Theta(NK^2)$
Residue additions	$\Theta(NK^2)$
Small-bit additions	$\Theta(NK)$
Precomputation storage (bits)	$\Theta(NK)$

Table 4 Performance of Bajard et al.'s ME algorithm [4,5] (without correction)

Operation	Total Count
Residue multiplications	$\Theta(NK^2)$
Residue additions	$\Theta(NK^2)$
Small-bit additions	0
Precomputation storage (bits)	$\Theta(NK)$

used in the performance tables is consistent with Table 1, with K being the number of moduli, $\hat{m}$ being the average of the modulli, and N being the word size of the operands. As can be seen, Phatak's RP-RNS approach, in theory, offers a useful tradeoff by requiring fewer expensive residue multiplications at the expense of additional precomputed constants.

The fastest implementation of RNS-based modular exponentiation to our knowledge is an optimized Cox-Rower implementation by Gandino, et al. [5] (2012) using their $\omega(c_i) \leq 3$ architecture. It executed an RNS Montgomery multiplication in 78 cycles with a cycle delay of 1.12, which approximates to 88 ns. In theory, by relying on additional precomputations, a hardware implementation of Phatak's QFS algorithm should be faster.

4 RP-RNS Hardware

We developed our implementation on the Xilinx Spartan-3E FPGA (XC3S500E) running with a 50 MHz clock. For design synthesis, we used the vendor's toolchain, Xilinx ISE v14.5. We did not create a gate-level VLSI design. To optimize execution time, we specified the connections between the arithmetic and memory components and left their instantiation to the vendor's synthesis tool. We used an FPGA for our implementation fabric because it is lower risk and cost compared to fabricating a custom ASIC.

4.1 Hardware Design Decisions

Phatak suggests a design that uses parallel-access storage and adder trees. This yields a design optimized for speed. We naively stored the lookup tables on the FPGA, which led us to minimizing logic utilization as our design strategy. In hindsight, we should have put the lookup tables in external memory.

Since the parallel-access storage and adder trees used a high number of logic cells, to allow more residue channels, we replaced them with equivalent sequential hardware. This carries a (relatively) hefty execution time penalty leading to an execution delay of $\mathrm{O}(N)$ (instead of $\mathrm{O}(\lg N)$).

Implementing the design on an FPGA provided the advantage that we could test our hardware design using arbitrary divisor sizes.

4.2 Hardware Architecture

Our hardware architecture comprises four components: the controller, the hardware channels, the redundant residue channel, and the fraction channel.

The controller contains logic for performing five operations: forward conversion, partial reconstruction, scaling, modular exponentiation, and reverse conversion.

A hardware channel executes residue operations for each residue channel – possibly multiple channels. We divided the moduli equally across all hardware channels such that each hardware channel supports at most $\lceil K/p \rceil$ residue channels where p is the number of hardware channels. Each hardware channel stores the lookup tables for its respective residue channels. The redundant residue channel is roughly identical to the hardware channels, but dedicated to the extra modulus m_e.

The fraction channel is similar to the redundant residue channel, except dedicated to the approximated fractional values. The channel's arithmetic logic must support word lengths of w_T bits where $w_T = w_I + w_F$. The value $w_I = \lceil \lg K \rceil$ supports the overflow of adding the truncated fractions. The value w_F is the fractional precision specified in the RP-RNS algorithms.

5 Testing Methodology

We conducted our tests using three apparatuses: synthesized hardware on the FPGA, hardware simulation using Xilinx iSim v14.5, and software simulation written in Python [14]. We numerically validated the functional correctness of each using 100 sets of random base, exponent, and divisor inputs for all divisor lengths between 4 and 20 bits. For the simulations, we were able to verify up to 256 bit divisors.

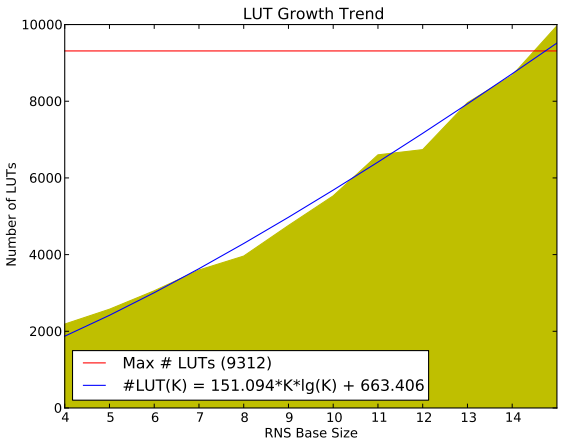

Fig. 1 FPGA logic utilization exhibits a superlinear growth as a function of the RNS base size when it should be approximately linear. We suspect the precomputed lookup tables (LUTs) are the factor.

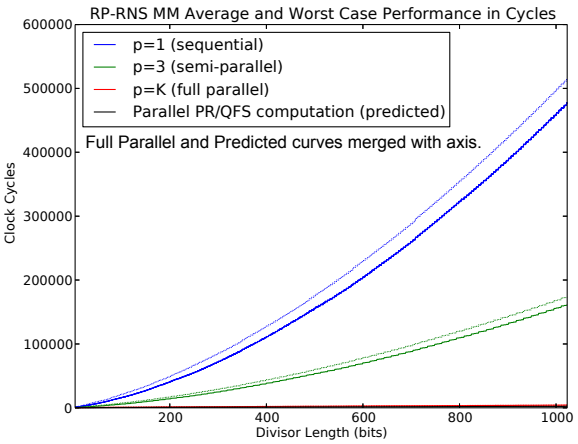

Fig. 2 A graphical summary of our cycle count data. Cycle counts for divisors up to 1024-bits using four varieties of parallelization.

The amount of testing we could conduct on the physical FPGA was limited. The synthesized hardware was limited to divisors no larger than 20 bits, which was a result of our naive decision to store the lookup tables on the FPGA.

6 Results

We collected metrics on FPGA logic utilization and execution time. We tracked only logic utilization for the synthesized hardware. We collected execution time as cycle counts using all three test apparatuses. The results here summarize data from Nguyen [8].

6.1 Logic Utilization

Figure 1 illustrates the dependence of logic utilization on RNS base size. We consider the utilization to depend solely on the size of the RNS base because the growth in moduli size is negligible ($\approx \lg\lg M$). For example, a 4096-bit divisor requires 16-bit moduli at most. We expected the growth to be linear in the RNS base size, but a regression using the Levenberg-Marquardt algorithm found in most statistical packages yielded a superlinear growth rate:

$$\#\mathrm{LUT}(K) \approx 151.094K\log(K) + 663.406. \quad (2)$$

Based on the toolchain's synthesis report, we believe our decision regarding the lookup table storage caused the superlinear growth rate. When we excluded the lookup tables, logic utilization grew at a rate approximately linear. Furthermore, the quantity of logic cells wasted for routing was negligible.

6.2 Execution Time

We tested our hardware using divisors between 20 and 1024 bits. In contrast to the literature, we did not optimize our implementation for a specific divisor nor did we optimize for speed. As such, our cycle count data applies only to our implementation and we feel our data are not suitable for direct comparison against existing implementations.

Figure 2 summarizes our cycle count data for computing a single multiplication modulo-D (a single scaling by a constant). The graph depicts four types of data pairs. Each pair includes average-case and worst-case cycle counts. The first three types represent different levels of parallelism: sequential, three hardware channels, and fully parallel. The fourth type is a theoretical modification that attempts to include Phatak's design decisions (see Section 4) into our software simulation of the hardware. We accomplished this by substituting the iterative table look-ups with a parallel table look-up in our software simulation and iterative adders with adder trees. For the average case data, the confidence interval around the mean is quite small. The 95% confidence interval around the mean was within ± 3 cycles for small divisors and no more than ± 10 cycles for 1024-bit divisors.

The sequential and semi-parallel implementations exhibit a cycle count that grows quadratically. This growth rate is due to our decision to forego Phatak's assumptions and our optimization choices. The full-parallel implementation, despite our design decisions, exhibits a trend that better reflects the $\mathrm{O}(\lg N)$ theoretical speed estimates (of a single modular multiplication) taking into account Phatak's design decisions (see Figure 3).

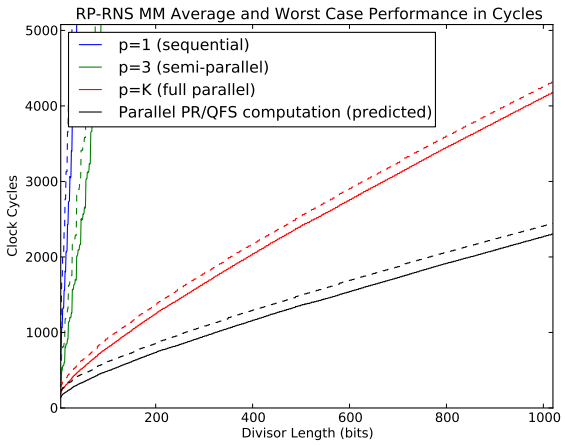

Fig. 3 Our hardware simulation shows that modifying our hardware to comply with Phatak's assumptions would enable us to execute in $O(\lg N)$ time for a single modular multiplication as promised by Phatak's analyses. (Close-up of Figure 2)

7 Conclusion

Using our implementation, we experimentally confirmed Phatak's performance analysis of the RP-RNS Quotient-First Scaling algorithm, which for a single modular multiplication is $O(\lg N)$ where N is the word length of the operands. Because QFS is performed in $O(N)$ iterations of the RP-RNS modular exponentiation algorithm, experimental confirmation of the RP-RNS scaling algorithm (QFS) gives strong evidence for the claimed $O(N \lg N)$ modular exponentiation run-time [12].

Though our implementation fails to outperform recently developed hardware such as discussed in Gandino, et al. [5], we believe lessons learned during our experience will be valuable for future implementations. Based on our preliminary experimental data and Phatak's analyses, we believe an implementation of the RP-RNS modular exponentiation algorithm, using all of Phatak's hardware assumptions and design decisions, will outperform implementations based on Montgomery's method.

As a new system, RP-RNS provides several open research problems. We highlight a few:

1. a speed-optimized implementation of RP-RNS modular exponentiation and repetition of our experiments;
2. a gate-level VLSI implementation of the RP-RNS modular exponentiation algorithm;
3. an ASIC implementation to determine if a custom circuit with all of Phatak's hardware assumptions can outperform other RNS algorithms;
4. a focused study on the partial reconstruction algorithm;
5. a method for selecting optimal RP-RNS bases.

Acknowledgements We thank Chintan Patel, Ryan Robucci, and David DeLatte for providing technical support and helpful comments on earlier drafts. Sherman and Nguyen were supported in part by the National Science Foundation under SFS grant 1241576.

References

1. Bajard, J.C., Didier, L.S., Kornerup, P.: An IWS Montgomery modular multiplication algorithm. In: Proceedings of the 13th Symposium on Computer Arithmetic (ARITH '97), ARITH '97, pp. 234–239. IEEE Computer Society, Washington, DC, USA (1997). URL http://dl.acm.org/citation.cfm?id=786448.786561
2. Bajard, J.C., Didier, L.S., Kornerup, P.: An RNS Montgomery modular multiplication algorithm. IEEE Trans. Comput. **47**(7), 766–776 (1998). DOI 10.1109/12.709376. URL http://dx.doi.org/10.1109/12.709376
3. Bajard, J.C., Didier, L.S., Kornerup, P.: Modular multiplication and base extensions in residue number systems. In: Proceedings of the 15th IEEE Symposium on Computer Arithmetic, ARITH '01, pp. 59–. IEEE Computer Society, Washington, DC, USA (2001). URL http://dl.acm.org/citation.cfm?id=872021.872466
4. Bajard, J.C., Imbert, L.: A full RNS implementation of RSA. IEEE Trans. Comput. **53**(6), 769–774 (2004). DOI 10.1109/TC.2004.2. URL http://dx.doi.org/10.1109/TC.2004.2
5. Gandino, F., Lamberti, F., Paravati, G., Bajard, J., Montuschi, P.: An algorithmic and architectural study on Montgomery exponentiation in RNS. Computers, IEEE Transactions on **61**(8), 1071–1083 (2012). DOI 10.1109/TC.2012.84
6. Kawamura, S., Koike, M., Sano, F., Shimbo, A.: Cox-Rower architecture for fast parallel Montgomery multiplication. In: Proceedings of the 19th international conference on Theory and application of cryptographic techniques, EUROCRYPT'00, pp. 523–538. Springer-Verlag, Berlin, Heidelberg (2000). URL http://dl.acm.org/citation.cfm?id=1756169.1756220
7. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**, 519–521 (1985)
8. Nguyen, C.D.: Fast modular exponentiation using residue domain representation: A hardware implementation and analysis. Master's thesis, University of Maryland, Baltimore County, CSEE Dept. (2013). URL http://pqdtopen.proquest.com/pubnum/1551346.html
9. Nozaki, H., Motoyama, M., Shimbo, A., Kawamura, S.i.: Implementation of RSA algorithm based on RNS Montgomery multiplication. In: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems, CHES '01, pp. 364–376. Springer-Verlag, London, UK, UK (2001). URL http://dl.acm.org/citation.cfm?id=648254.752572
10. Phatak, D.S.: RNS-ARDSP: A novel, fast residue number system using approximate rational domain scaled precomputations, Part I RP-PR: Reduced precision partial reconstruction and its application to fast RNS base extensions and/or base-changes. Tech. Rep. TR-CS-10-01, University of Maryland, Baltimore County, CSEE Dept. (2010). UMBC Technical Report TR-CS-10-01. Revised: November, 2013
11. Phatak, D.S.: RNS-ARDSP: A novel, fast residue number system using approximate rational domain scaled precomputations, Part II: SODIS (sign and overflow detection by interval separation) algorithm, the quotient first scaling/division-by-a-constant algorithm and their applications to expedite modular reduction and modular exponentiation. Tech. Rep. TR-CS-10-02, University of Maryland, Baltimore County, CSEE Dept. (2010). UMBC Technical Report TR-CS-10-02. Revised: November, 2013
12. Phatak, D.S.: RNS-ARDSP: A novel, fast residue number system using approximate rational domain scaled precomputations, Part III: Fast convergence division algorithms and a broad overview of

other potential enhancements to RNS and their applications. Tech. Rep. TR-CS-10-03, University of Maryland, Baltimore County, CSEE Dept. (2010). UMBC Technical Report TR-CS-10-03. Revised: November, 2013
13. Phatak, D.S.: Residue number systems methods and apparatuses. Patent Application (2011). US 2011/0231465 A1
14. van Rossum, G., Drake, F.L.: The Python Language Reference Manual. Network Theory Ltd. (2011)
15. Shenoy, A., Kumaresan, R.: Residue to binary conversion for RNS arithmetic using only modular look-up tables. Circuits and Systems, IEEE Transactions on **35**(9), 1158–1162 (1988). DOI 10.1109/31.7577
16. Shenoy, A., Kumaresan, R.: Fast base extension using a redundant modulus in RNS. Computers, IEEE Transactions on **38**(2), 292–297 (1989). DOI 10.1109/12.16508

A RP-RNS Quotient-First Scaling Algorithm

This appendix states the RP-RNS quotient-first scaling algorithm for reference purposes. See Phatak [10–13] for explanations and discussions. Though the following design features have appeared separately in other RNS algorithms, the novelty emerges in how RP-RNS combines these features and the speed gained:

- the usage of approximation [6];
- the 1- or 2-bit redundant modulus [15,16];
- the usage of rational equations over integer equations [6];
- the precomputed lookup tables [1–3,6,15,16]; and
- the choice of many small moduli as the RNS base.

The precision of the approximations stored in the lookup tables is chosen to bound the error such that the final result is one of two candidates. Recently, it has come to our attention that Phatak's sign-detection algorithm contains an error. Since we did not implement the final sign-detection step (used to choose among the two candidates), our implementation is unaffected.

The QFS algorithm is based on the following rational form of the division algorithm:

$$Q = \frac{Z}{D} - \frac{R}{D}. \tag{3}$$

The algorithm determines a quotient estimate, $\hat{Q}$, of Z divided by D. Despite the fact that the output of the QFS algorithm is not always exact, this poses no problem for the RP-RNS modular exponentiation algorithm as noted by Phatak [11].

The QFS algorithm uses the RP-RNS partial reconstruction algorithm which is described in detail in [10]. The partial reconstruction algorithm is based on the following form of the Chinese Remainder Theorem:

$$Z = \sum_{i=1}^{K} \rho_i M_i - Rc_Z M. \tag{4}$$

Denoting the summation in Equation 4 as Z_T, Phatak defines partial reconstruction as computation of the reconstruction coefficient Rc_Z without fully computing Z_T.

The QFS algorithm accepts as its input a number Z in RP-RNS form and outputs the quotient estimate $\hat{Q}$ in RP-RNS form and an exactness flag. The algorithm asserts the flag only if $\hat{Q} = Q$. The QFS algorithm uses two lookup tables. The first lookup table uses the ρ values from the partial reconstruction algorithm as its indices. The second lookup table uses Rc_Z as its index. The equations for the lookup table entries are

$$\begin{aligned} Q_i &= \left\lfloor \tfrac{M_i \rho_i}{D} \right\rfloor \qquad & R_i &= \left\lfloor \tfrac{M_i \rho_i - Q_r D}{D} 2^{w_F} \right\rfloor \\ Q_{rc} &= \left\lfloor \tfrac{M Rc_Z}{D} \right\rfloor \qquad & R_{rc} &= \left\lceil \tfrac{M Rc_Z - Q_{rc} D}{D} 2^{w_F} \right\rceil. \end{aligned} \tag{5}$$

The QFS algorithm invokes the partial reconstruction algorithm to calculate the lookup table indices. The Q_i and Q_{rc} values are the integer part of the quotient. The R_i and R_{rc} values are the approximated fractional part of the quotient. The chosen rounding modes lead to a quotient underestimate, similar to the partial reconstruction algorithm. The fractions are truncated to w_F bits where $w_F \geq \lceil \lg K + 1 \rceil$. Together the tables require $\mathrm{O}\left(N^3 \lg\lg N\right)$ bits of storage.

Algorithm 1: RP-RNS QFS Quotient Estimation

Input : An integer z in RP-RNS form
Output: $\hat{Q}$ in RP-RNS form and the exactness flag QExact. $\hat{Q}$ is an approximation to $Q = z/D \mod N$, where D is a pre-defined fixed divisor. If QExact = True, then $\hat{Q} = Q$. Otherwise, if QExact = False, then either $\hat{Q} = Q$ or $\hat{Q} = Q + D$.

begin
  /* The QFS lookup tables have multiple columns. Though we identify the columns, we omit the structure and refer the reader to [13, 11]. The lookup tables are populated based on the fixed divisor D. The call to PartialReconstruction is used to determine the reconstruction coefficient and is described in detail by Phatak [10]. */
  $Rc_z, (\rho_1, \rho_2, \ldots, \rho_K), n \leftarrow \texttt{PartialReconstruction}(z)$;
  /* Quotient integer part computation. */
  $\overline{Q_i} \leftarrow \texttt{QuotientTable1}(i, \rho_i, 5)$ for $1 \leq i \leq K$;
  $\overline{Q_{Rc_z}} \leftarrow \texttt{QuotientTable2}(Rc_z, 4)$;
  $\overline{Q} \leftarrow \sum_{i=1}^{K} \overline{Q_i} - \overline{Q_{Rc_z}}$;
  $[Q_i]_{m_e} \leftarrow \texttt{QuotientTable1}(i, \rho_i, 4)$ for $1 \leq i \leq K$;
  $[Q_{Rc_z}]_{m_e} \leftarrow \texttt{QuotientTable2}(Rc_z, 3)$;
  $[Q]_{m_e} \leftarrow \sum_{i=1}^{K} [Q_i]_{m_e} - [Q_{Rc_z}]_{m_e}$;
  /* Quotient fractional part computation. */
  $R_i \leftarrow \texttt{QuotientTable1}(i, \rho_i, 6)$;
  $R_{Rc_z} \leftarrow \texttt{QuotientTable2}(Rc_z, 5)$;
  $Q_f \leftarrow \sum_{i=1}^{K} R_i - R_{Rc_z}$;
  /* Quotient exactness assessment. */
  $Q_L \leftarrow \texttt{RightShift}(Q_f, w_F)$;
  $Q_H \leftarrow \texttt{RightShift}(Q_f + n, w_F)$;
  $\texttt{QExact} \leftarrow (Q_L == Q_H)$;
  /* Combine to form the quotient estimate. */
  $\overline{\hat{Q}} \leftarrow \overline{Q} + \overline{Q_L}$;
  $[\hat{Q}]_{m_e} \leftarrow [Q]_{m_e} + [Q_L]_{m_e}$;
  return $\left(\overline{\hat{Q}}, [\hat{Q}]_{m_e}, \texttt{QExact}\right)$;