A Optimized Additive Overflow Detection Algorithm for the Generalized Residue Number System

1 Introduction

The Residue Number System (RNS) provides a way to perform arithmetic operations on large integers efficiently in terms of space and time, which are commonplace in cryptography. Given a set of co-prime numbers $\{m_1, m_2, \ldots, m_n\}$, called a modulus set, it follows by the Chinese Remainder Theorem that any number $X < m_1 m_2 \ldots m_n$ can be represented uniquely as the set of residues of X modulo each of the m_i. This set of residues becomes the representation of X in the RNS defined by the modulus set $\{m_1, m_2, \ldots, m_n\}$. Performing arithmetic operations in a RNS decomposes a single operation with large integers into multiple independent operations involving smaller integers, which may be performed in parallel.

A significant issue when performing addition within a RNS is additive overflow detection. Additive overflow occurs when the sum of two numbers cannot be represented in the RNS, due to the sum exceeding the modulus product $M = m_1 m_2 \ldots m_n$. One way of overcoming this problem is through parity checking. Within this paper, we compare the performance of several algorithms for parity checking in an RNS: a brute force algorithm, an algorithm published in Mi and Chiang (1992), and an optimized algorithm that we constructed. While Mi and Chiang's approach outperforms the brute force algorithm, our algorithm performs best overall.

2 Related Work

Mi Lu and Jen-Shuin Chiang (1992) propose an approach to parity checking that still appears to be the state-of-the-art for generalized residue number systems. Let $\{m_1, m_2, ..., m_n\}$ be any collection of pair-wise relatively prime, odd integers, with $M = \prod_{i=1}^{n} m_i$, and $\hat{m}_i = \frac{M}{m_i}$, $\forall i \in \{1, 2, ..., n\}$. For any two non-negative integers X,Y $\in [0, M)$, with respective residue representations $\{x_1, x_2, ..., x_n\}$ and $\{y_1, y_2, ..., y_n\}$, we express the residue representation of the sum, Z = (X+Y), by the collection of residues $\{z_1, z_2, ..., z_n\}$, where $z_i = |x_i + y_i|_{m_i}, \forall i \in \{1, 2, ..., n\}$. Understanding that we maintain the true parity of the sum as the residue of the redundant modulus two, we need only to determine the parity of the sum within the original RNS.

By the Chinese Remainder Theorem, we understand that we may reconstruct the decimal representation of our integer through the following formula:

$$|Z|_M = |\sum_{i=1}^{n} \hat{m}_i |\frac{z_i}{\hat{m}_i}|_{m_i}|_M \tag{1}$$

Since $|Z|_M$ denotes the least positive residue of Z modulo M, there exists a distinct, non-negative integer 'r' such that:

$$|Z|_M = \sum_{i=1}^{n} \hat{m}_i |\frac{z_i}{\hat{m}_i}|_{m_i} - rM \tag{2}$$

We have constructed our modulus set to exclusively maintain elements of odd parity. Therefore, we understand that M is odd and so $\hat{m}_i$ is odd for all $i \in \{1, 2, ..., n\}$. Hence, the parity of $|Z|_M$ may easily be determined by the following relation:

$$\mathcal{P}(|Z|_M) = \mathcal{P}(|\frac{z_1}{\hat{m}_1}|_{m_1}) \oplus \mathcal{P}(|\frac{z_2}{\hat{m}_2}|_{m_2}) \oplus \cdots \oplus \mathcal{P}(|\frac{z_n}{\hat{m}_n}|_{m_n}) \oplus \mathcal{P}(r) \tag{3}$$

We may pre-calculate and store the values $|\frac{z_i}{\hat{m_i}}|_{m_i}, \forall i \in \{1, 2, ..., n\}$, as they are sufficiently small. Therefore, to accuractely calculate the parity of $|Z|_M$, we now simply need to determine the parity of the integer 'r'. Let $S_i = |\frac{z_i}{\hat{m_i}}|_{m_i}, \forall i \in \{1, 2, ..., n\}$. Then, rewriting equation (2), we understand that we may determine 'r' as follows:

$$r = \lfloor \sum_{i=1}^{n} \frac{S_i}{m_i} \rfloor \tag{4}$$

Mi Lu & Jen-Shiun Chiange argue that accurately calculating the positive integer 'r' may be accomplished through an approximation of the values $\frac{S_i}{m_i}$ for all $i \in \{1, 2, \cdots, n\}$. Their approximations require the use of 't' bits, where $t > \lceil log_2(nM) \rceil$. We claim that such a calculation may be accomplished with fewer bits, and - despite the recommended use of a lookup table - the large size of these calculations increase the necessary time of program execution. We aim to increase the efficiency of this algorithm by reducing the space required for each calculation.

3 Methodology

We define our RNS by the collection of relatively prime moduli of odd parity, $\{m_1, m_2, ..., m_n\}$. For any non-negative integer $X \in [0, M)$, with residue representation $\{x_1, x_2, ..., x_n\}$, we calculate the parity of X by the relation:

$$\mathcal{P}(|X|_M) = \mathcal{P}(|\frac{x_1}{\hat{m_1}}|_{m_1}) \oplus \mathcal{P}(|\frac{x_2}{\hat{m_2}}|_{m_2}) \oplus \cdots \oplus \mathcal{P}(|\frac{x_n}{\hat{m_n}}|_{m_n}) \oplus \mathcal{P}(r) \tag{5}$$

Once again, understanding that the values $|\frac{x_i}{\hat{m_i}}|_{m_i}$ are sufficiently small for all $i \in \{1, 2, ..., n\}$, they are mutually calculated and stored for future computation. Therefore, by the relation defined above, we may easily determine the parity of our integer X by accurately calculating the non-negative integer 'r' and performing an exclusive-or operation with its parity. We determine this integer value through the relation:

$$r = \lfloor \sum_{i=1}^{n} \frac{|\frac{x_i}{\hat{m_i}}|_{m_i}}{m_i} \rfloor \tag{6}$$

Let $S_i = |\frac{x_i}{\hat{m_i}}|_{m_i}$ and $u_i = \frac{S_i}{m_i}$ for all $i \in \{1, 2, \cdots, n\}$ Our algorithm modification provides a reduction in spatial complexity by minimizing the number of bits required to determine the integer value 'r'. Let P be the largest element of our modulus set, and define the positive integer $\hat{m}_P = \prod_{m_i \neq P} m_i$. We construct the active ranges, $U, V \subset [0, M)$, such that

$$U = [0, \frac{(P-1)M}{P}] \tag{7}$$

$$V = [\frac{M}{P}, M) \tag{8}$$

Clearly, we understand that $U \cup V = [0, M)$. Let $X \in [0, M)$. Then, we consider the following cases:

1. Suppose $X \in U$. For all $i \in \{1, 2, .., n\}$, we define the value $\hat{u_i} = \frac{\lceil 2^t u_i \rceil}{2^t}$ to be the approximation of the value u_i. Then, we must find some non-negative integer 't' such that:

$$r = \lfloor \sum_{i=1}^{n} \hat{u_i} \rfloor \tag{9}$$

We understand that this approximation method adds error to the initial sum. Therefore, for all $i \in \{1, 2, ..., n\}$, there exists a positive error e_i such that $\hat{u_i} = u_i + e_i$. Then, we understand that:

$$\lfloor \sum_{i=1}^{n} \hat{u_i} \rfloor = \lfloor \sum_{i=1}^{n} u_i + \sum_{i=1}^{n} e_i \rfloor \tag{10}$$

$$= \lfloor (r + \frac{X}{M}) + \sum_{i=1}^{n} e_i \rfloor \tag{11}$$

Therefore, understanding that $X \in [0, \frac{(P-1)M}{P}]$, we calculate the upper-bound of each error value e_i as follows:

$$\frac{X}{M} + \sum_{i=1}^{n} e_i < 1 \tag{12}$$

$$\sum_{i=1}^{n} e_i < 1 - \frac{X}{M} \tag{13}$$

$$\sum_{i=i}^{n} e_i < 1 - \frac{\frac{(P-1)M}{P}}{M} \tag{14}$$

$$\sum_{i=i}^{n} e_i < 1 - \frac{(P-1)}{P} \tag{15}$$

$$\sum_{i=i}^{n} e_i < \frac{1}{P} \tag{16}$$

$$e_i < \frac{1}{nP} \tag{17}$$

Hence, if $X \in U$, then we may accurately determine the non-negative integer 'r' by choosing $t > \lceil log_2(nP) \rceil$.

2. Suppose $X \in V$. For all $i \in \{1, 2, .., n\}$, we define the value $\hat{v_i} = \frac{\lfloor 2^t u_i \rfloor}{2^t}$ to be the approximation of the value u_i. Then, we must find some non-negative integer 't' such that:

$$r = \lfloor \sum_{i=1}^{n} \hat{v_i} \rfloor \tag{18}$$

We understand that this approximation method subtracts error from the initial sum. Therefore, for all $i \in \{1, 2, ..., n\}$, there exists a positive error e_i such that $\hat{v_i} = u_i - e_i$. Then, by equation (4), we understand that:

$$\lfloor \sum_{i=1}^{n} \hat{v_i} \rfloor = \lfloor \sum_{i=1}^{n} u_i - \sum_{i=1}^{n} e_i \rfloor \tag{19}$$

$$= \lfloor (r + \frac{X}{M}) - \sum_{i=1}^{n} e_i \rfloor \tag{20}$$

Therefore, understanding that $X \in [\frac{M}{B}, M)$, we determine the upper-bound of each value $e_i, \forall i \in \{1, 2, ..., n\}$ as follows:

$$\frac{X}{M} - \sum_{i=1}^{n} e_i > 0 \tag{21}$$

$$\sum_{i=1}^{n} e_i < \frac{X}{M} \tag{22}$$

$$\sum_{i=i}^{n} e_i < \frac{(\frac{M}{P})}{M} \tag{23}$$

$$\sum_{i=i}^{n} e_i < \frac{1}{P} \tag{24}$$

$$e_i < \frac{1}{nP} \tag{25}$$

Now, we ignore the case where X is zero, as its value may clearly be stored within any positive number of bits. Hence, if $X \in V$, then we may accurately determine the non-negative integer 'r' by choosing $t > \lceil log_2(nP) \rceil$.

Therefore, if $X \in [0, M)$, then we may accurately calculate the non-negative integer 'r' by choosing $t > \lceil log_2(nP) \rceil$. We cannot efficiently determine whether $X \in U$ or $X \in V$, as such information would require an integer comparison, which is computationally infeasible within the RNS. Regardless, such information is not necessary to accurately determine the parity of X. We make the assumption that $X \in U$, and so determine the approximation $\hat{u_i}$, $\forall i \in \{1, 2, ..., n\}$. Similarly, we make the assumption that $X \in V$, and so determine the approximation $\hat{v_i}$, $\forall i \in \{1, 2, ..., n\}$. Understanding that $U \cup V = [0, M)$, it clearly follows that either $X \in U$ or $X \in V$. Therefore, $r = \lfloor \sum_{i=1}^{n} \hat{u_i} \rfloor$ or $\lfloor \sum_{i=1}^{n} \hat{v_i} \rfloor$. Consider the following cases:

1. Suppose $\lfloor \sum_{i=1}^{n} \hat{u_i} \rfloor = \lfloor \sum_{i=1}^{n} \hat{v_i} \rfloor$. Then, we understand that either $X \in U \cap V$, or the the sum of approximation errors is less than our upper-bound. In either case, since we have that $r = \lfloor \sum_{i=1}^{n} \hat{u_i} \rfloor$ or $\lfloor \sum_{i=1}^{n} \hat{v_i} \rfloor$, we may determine the parity of X as follows:

$$\mathcal{P}(X) = \mathcal{P}(|\frac{x_1}{\hat{m_1}}|_{m_1}) \oplus \mathcal{P}(|\frac{x_2}{\hat{m_2}}|_{m_2}) \oplus \cdots \oplus \mathcal{P}(|\frac{x_n}{\hat{m_n}}|_{m_n}) \oplus \mathcal{P}(\lfloor \sum_{i=1}^{n} \hat{u_i} \rfloor) \tag{26}$$

Therefore, if $\lfloor \sum_{i=1}^{n} \hat{u_i} \rfloor = \lfloor \sum_{i=1}^{n} \hat{v_i} \rfloor$, then we are done.

2. Suppose $\lfloor \sum_{i=1}^{n} \hat{u_i} \rfloor \neq \lfloor \sum_{i=1}^{n} \hat{v_i} \rfloor$. Then, we maintain that $X \in \overline{U \cap V} = [0, \frac{M}{P}) \cup (\frac{(P-1)M}{P}, M)$. We understand that $X \in [0, \hat{m_P}) \cup (\hat{m_P}(P-1), M)$ and so

$$X = |X|_{\hat{m_P}} + s\hat{m_P} \tag{27}$$

where $s \in \{0, (P-1)\}$. Since P and $\hat{m_P}$ are odd integers, this implies that:

$$\mathcal{P}(X) = \mathcal{P}(|X|_{\hat{m_P}}) \tag{28}$$

Let Q be the largest element of the reduced modulus set $\mathcal{M}_P = \{m_1, m_2, \cdots, m_n\} \setminus \{P\}$. Equivalently, we shall choose to express $\mathcal{M}_P$ by the set $\{m_{P_1}, m_{P_2}, \cdots, m_{P_{n-1}}\}$. Define $\hat{m_Q} = \frac{\hat{m_P}}{Q}$ and construct the values $\hat{m}_{P_i}$ by $\hat{m}_{P_i} = \frac{\hat{m}_i}{P}$. Then, we understand that $|X|_{\hat{m}_p}$ maintains a

distinct residue representation within $\mathcal{M}_P$. Let $\{x_{P_1}, x_{P_2}, \cdots, x_{P_{n-1}}\}$ represent these residues. Then, by the Chinese Remainder Theorem, it follows that:

$$|X|_{\hat{m_B}} = |\sum_{i=1}^{n-1} \hat{m_{P_i}} |\frac{x_{P_i}}{m_{P_i}}|_{m_{P_i}}|_{\hat{m_P}} \tag{29}$$

$$= \sum_{i=1}^{n} \hat{m_{P_i}} |\frac{x_{P_i}}{m_{P_i}}|_{m_{P_i}} - r_2 \hat{m}_P \tag{30}$$

Therefore, we may determine the parity of X as follows:

$$\mathcal{P}(X) = \mathcal{P}(|\frac{x_{P_1}}{\hat{m_{P_1}}}|_{m_{P_1}}) \oplus \mathcal{P}(|\frac{x_{P_2}}{\hat{m_{P_2}}}|_{m_{P_2}}) \oplus \cdots \oplus \mathcal{P}(|\frac{x_{P_{n-1}}}{\hat{m_{P_{n-1}}}}|_{m_{P_{n-1}}}) \oplus \mathcal{P}(r_2) \tag{31}$$

Hence, we have reduced the original parity calculation to a more manageable subproblem. Through comparing approximation procedures, we must either accurately determine the parity of our integer with a sufficiently small number of bits, or we may reduce the problem of parity calculation to a more manageable subproblem. Further, we may execute this algorithm iteratively until the parity of the integer is found, or until a trivial subproblem is reached. When utilizing a lookup table, we make the assumption that our approximated values $\hat{u}_i$ and $\hat{v}_i$ have been precalculated and stored within a table for all $i \in \{1, 2, \cdots, n\}$. As such, when utilizing a lookup table, we merely retrieve these associated values from the table, sum them together and compare them with each iteration.

4 Complexity Analysis

We understand that each algorithm we analyze is dominated by multiplication and division operations. Understanding that multiplication and division maintain the same complexity, we denote this complexity by $\mathbb{M}(k)$, where k is the number of bits within the operation. Consider the following:

1. We understand that our brute force procedure utilizes the following formula in its calculations:

$$|X|_M = |\sum_{i=1}^{n} \hat{m_i} |\frac{x_i}{\hat{m_i}}|_{m_i}|_M \tag{32}$$

Clearly, we understand that our values $\hat{m}_i$ are bounded above by M for all $i \in \{1, 2, \cdots, n\}$. Therefore, understanding that we maintain $n-1$ multiplicative procedures on inputs of size at most $\lceil \log M \rceil$, and that this procedure dominates the runtime of our algorithm, the brute force algorithm complexity is bounded above by:

$$T(n, M) \in O(n \cdot \mathbb{M}(\lceil \log M \rceil)) \tag{33}$$

Additionally, since we maintain that the brute force algorithm does not permanently store any value, the spatial complexity of our brute force algorithm is within:

$$S(n, M) \in O(\lceil \log M \rceil) \tag{34}$$

2. The time complexity of Mi Lu & Jen-Shiun Chiangs' algorithm is dominated by the calculation of the integer value 'r'. As expressed within this algorithm, $r = \lfloor \sum_{i=1}^{n} \frac{S_i}{m_i} \rfloor$. Clearly, the construction of this integer is dominated by the division operations $\frac{S_i}{m_i}$ for all $i \in \{1, 2, \cdots, n\}$.

Understanding that there are 'n' operations, each operating on $\lceil \log(nM) \rceil$ bits, this implies that the complexity of their algorithm is bounded above by the following:

$$T(n, M) \in O(n \cdot \mathbb{M}(\lceil \log(nM) \rceil)) \tag{35}$$

When excluding a lookup table, we understand that no values are permanently stored. Therefore, it follows that the spatial complexity of their algorithm lies within:

$$S(n, M) \in O(\lceil \ log(nM) \rceil) \tag{36}$$

When utilizing Mi Lu & Jen-Shiun Chiang's algorithm with the incorporation of a lookup table, they make the assumption that the approximated values $\hat{u}_i$ are precalculated and stored for all $i \in \{1, 2, \cdots, n\}$. Therefore, in calculating the integer value 'r', we understand that the time complexity of their algorithm is simply bounded above by the cost of addition. Therefore, the table implementation of their algorithm maintains the following time complexity:

$$T(n, M) \in O(n \cdot \lceil \log(nM) \rceil) \tag{37}$$

Additionally, allowing P to be largest modulus within the collection, the spatial complexity of the algorithm is bounded above by the size of the lookup table. Hence, the spatial complexity of the lookup table implementation of their algorithm lies within:

$$S(n, M) \in O(n \cdot P \lceil \log(nM) \rceil) \tag{38}$$

3. As with Mi Lu & Jen-Shiun Chiang's parity calculation algorithm, each iteration of our algorithm is dominated by the calculation of the integer value 'r'. We have previously expressed that each operation within our algorithm requires at most $\lceil \log nP \rceil$ bits, where P is the largest modulus within our collection. Understanding that we may have at most 'n' iterations of our algorithm, this implies the following upper bound on the time complexity where we forbid the construction of a lookup table:

$$T(n, M) \in O(n^2 \cdot \mathbb{M}(\lceil \log nP \rceil)) \tag{39}$$

We also maintain the following spatial complexity when excluding a lookup table:

$$S(n, M) \in O(\lceil \log nP \rceil) \tag{40}$$

Additionally, if we permit the use of a lookup table, then the complexity of our algorithm is simply bounded above by the cost of addition. We determine this complexity to be the following:

$$T(n, M) \in \Theta(n^2 \lceil \log nP \rceil) \tag{41}$$

Likewise, we determine the spatial complexity of our algorithm to be bounded by the size of the table, which is bounded above by the following:

$$S(n, M) \in O(n^2 \cdot P \lceil \log nP \rceil) \tag{42}$$

5 Experimental Setup

We performed our experiments on a Red Hat Enterprise Linux Server 6.5 (Santiago). This server maintains two Intel Xeon E5645 processors and 64GB of RAM. Each Intel Xeon Processor E5645 maintains six cores, and twelve threads, with a 12MB L2 cache and 2.4GHz system clock speed. Each sample of code within our experiment was written in C++ and compiled with GCC 4.4.7. Additionally, we wrote each algorithm to employ the use of the GNU Multiple Precision Arithmetic Library (GMP 5.1.3), and all execution times were recorded utilizing the standard *time spec* structure's nanosecond time scale and real time argument.

We construct our modulus sets as a collection of at most one-hundred relatively-prime positive integer values, with each modulus within the set being the product of at most ten prime numbers, randomly selected from a collection of the first one-thousand odd primes. We performed our experiments on a collection of one-thousand randomly generated modulus sets. Each modulus set maintained an associated residue collection, which was randomly generated to provide an average case analysis of each algorithm.

All table implementations were assumed to have an instant access time, and table values were constructed and stored within an array before execution times were recorded. We likewise do not consider the time necessary to construct these tables, as every component native to the RNS algorithms is assumed to be precomputed. Each measurement of execution time directly corresponds to all operations performed on a given residue set.

Our results were plotted using Matlab R2013a. Understanding that Matlab's input values are bounded by the standard integer and floating point representations, our expressible range of modulus products are bounded above by 10^{307}. Such computations are considered small in consideration to many common RNS implementations. However, we could not express such results within any open source plotting software. As such, our execution times are limited to microseconds.

6 Results

We have shown that with a small change to the Lu-Chiang algorithm, you can obtain a statistically significant improvement to the runtime and asymptotic growth analysis. Lu-Chiang's contribution was that you can use a table to do some of the calculations before hand. However, their approach performs worse than brute force without the use of a table. With our modifications, we were able to achieve comparable runtime to brute force and halved the runtime of the worst case. With a table, our performance was unmatched. Our worst case was 17% of brute force and 35% of Lu-Chiang. Our average case was less than half of Lu-Chiang, and a quarter of brute force with a tight standard deviation. These results are presented below.

	Brute Force	Mi Lu & Jen-Shiun	Ours
Average Case	4290.18	6200.67	4777.60
Std Deviation	2723.03	3361.79	2278.77
Worst Case	34467.00	24239.00	14888.00

Table 1: Comparison of run-time analysis for all algorithms with a non-table based implementation.

	Brute Force	Mi Lu & Jen-Shiun	Ours
Average Case	4290.18	2069.31	960.05
Std Deviation	2723.03	1144.32	387.34
Worst Case	34467.00	17497.00	6194.00

Table 2: Comparison of run-time analysis for all algorithms with a table based implementation.

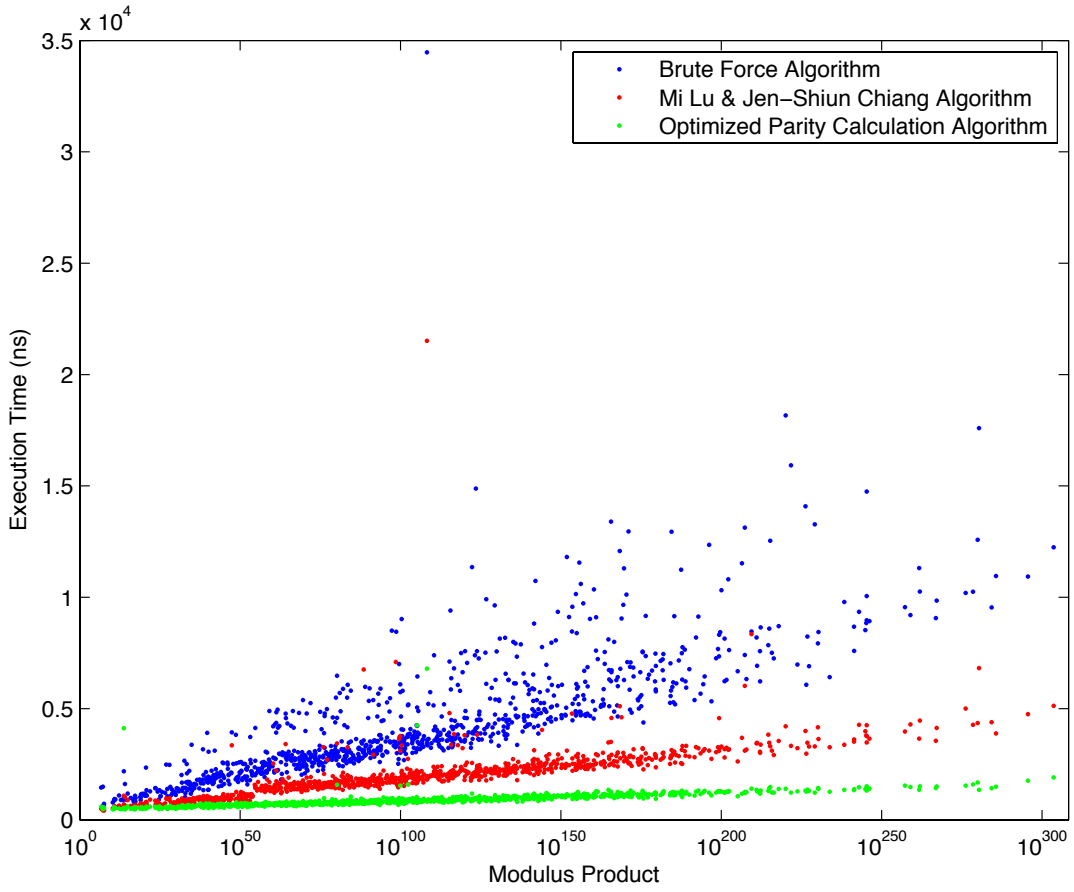

Figure 1: Comparing all three algorithms. Mi Lu and Optimized are using their table implementations.

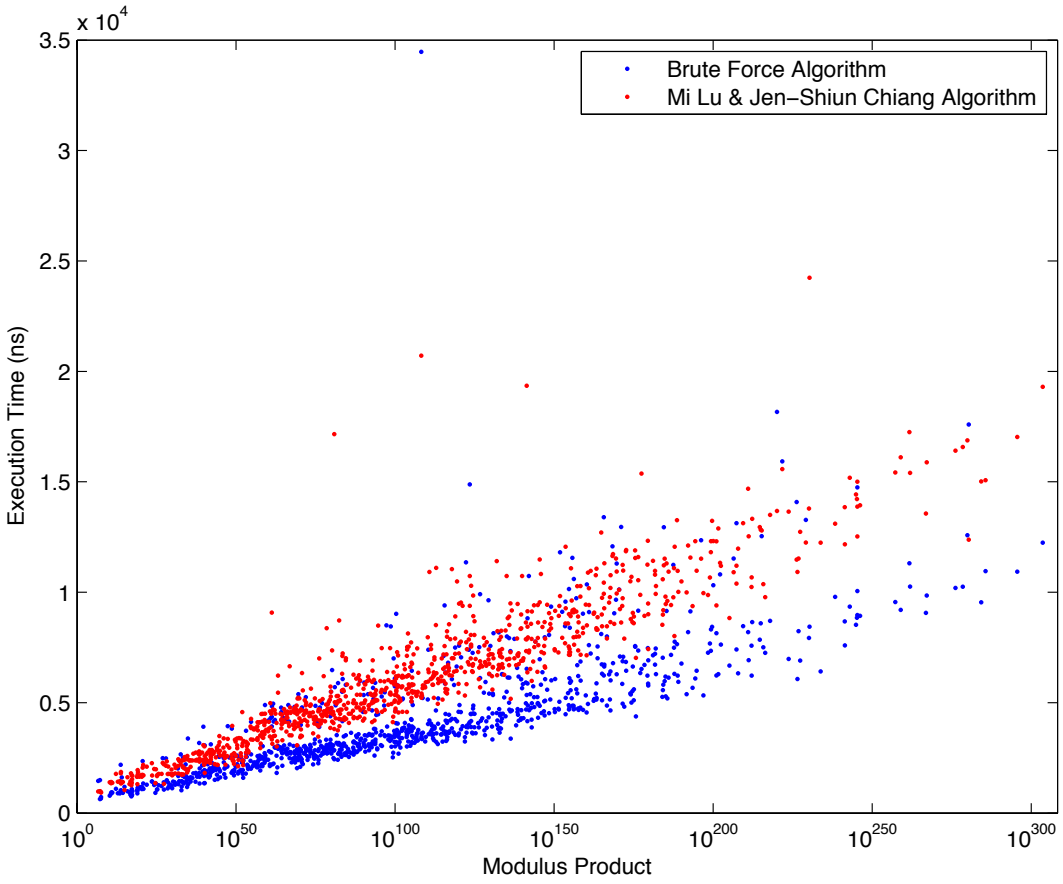

Figure 2: Comparing Brute Force to the Mi Lu algorithm without a table.

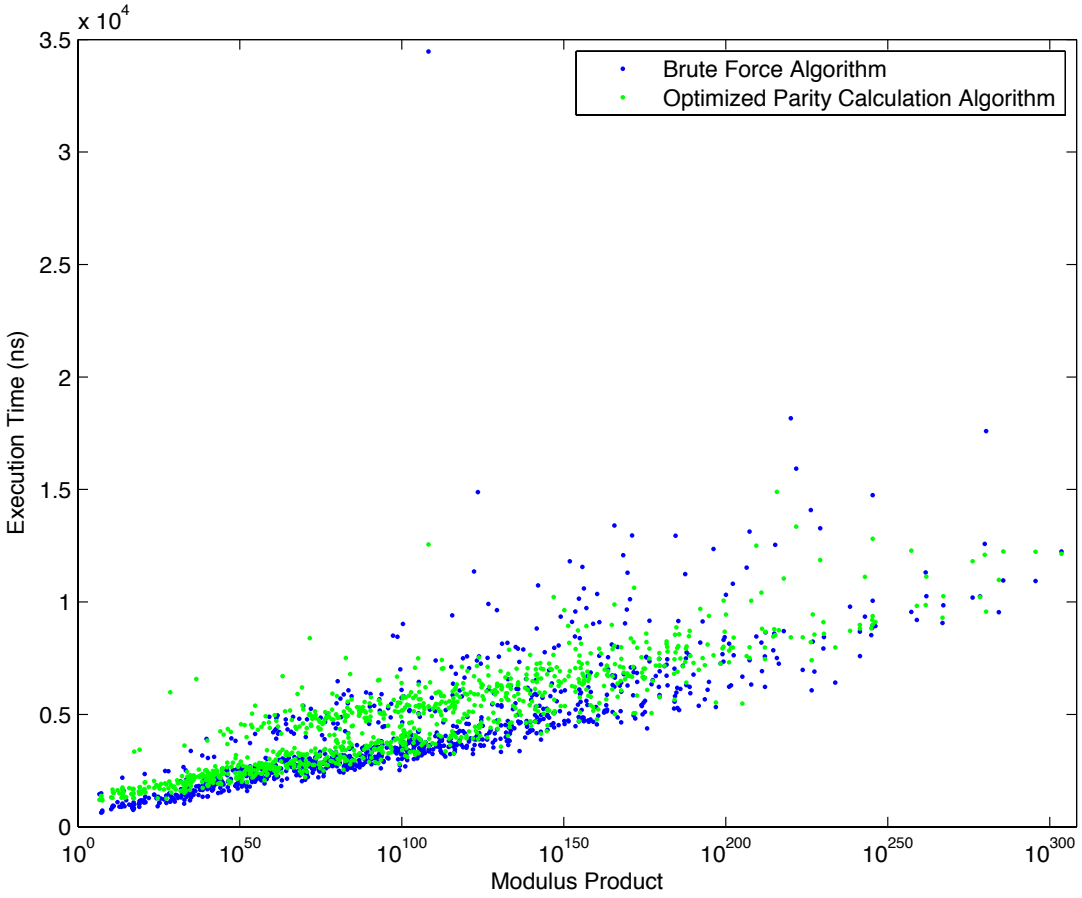

Figure 3: Brute Force compared to Optimized without table.

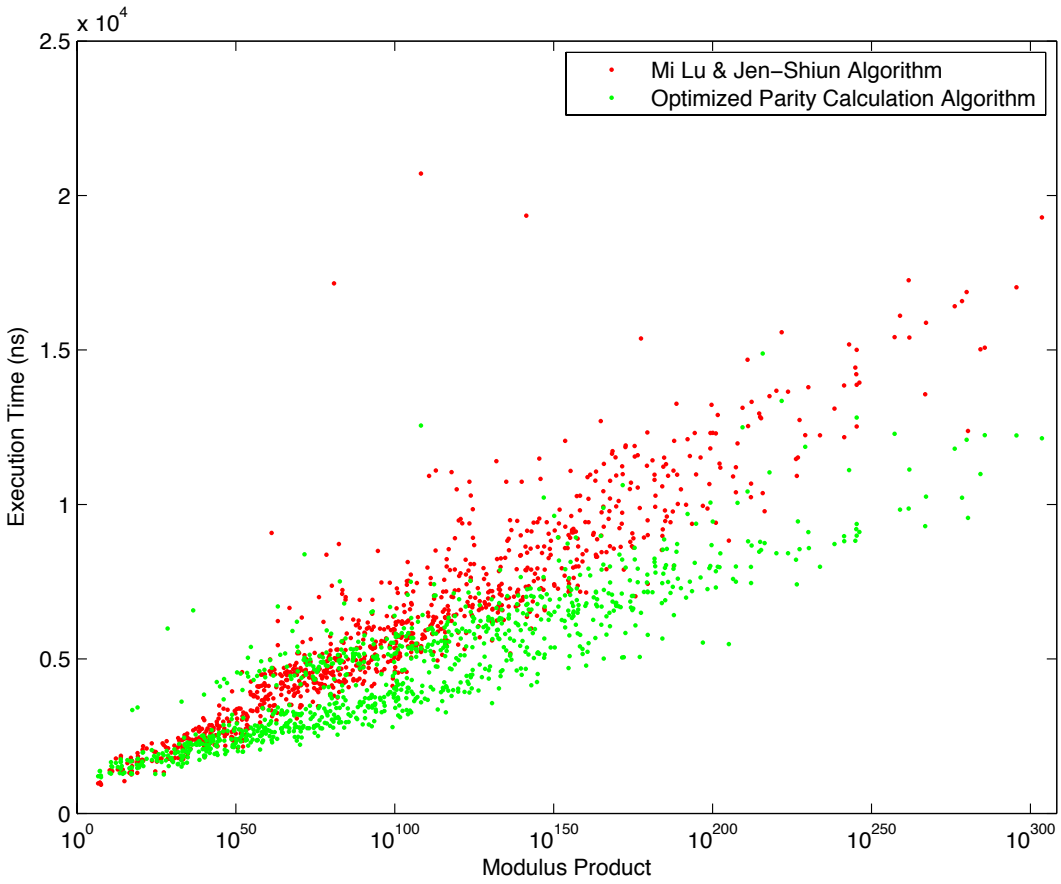

Figure 4: Mi Lu compared to Optimized, both without tables.

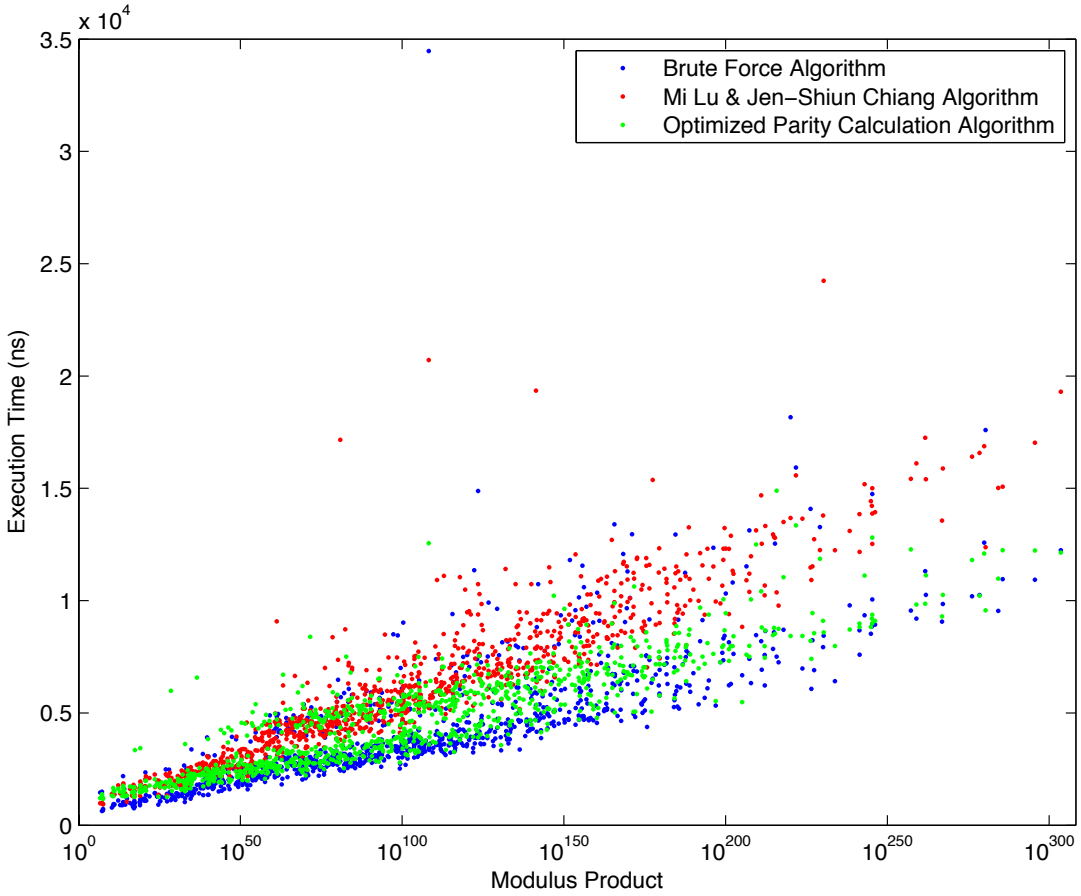

Figure 5: Comparing all three algorithms. Mi Lu and Optimized are using their implementations without tables.