Accelerating Cryptanalysis with the Method of Four Russians

Gregory V. Bard[*]

July 22, 2006

Abstract

Solving a dense linear system of boolean equations is the final step of several cryptanalytic attacks. Examples include stream cipher cryptanalysis via XL and related algorithms, integer factorization, and attacks on the HFE public-key cryptosystem. While both Gaussian Elimination and Strassen's Algorithm have been proposed as methods, this paper specifies an algorithm that is much faster than both in practice. Performance is formally modeled, and experimental running times are provided, including for the optimal setting of the algorithm's parameter. The consequences for published attacks on systems are also provided. The algorithm is named Method of Four Russians for Inversion (M4RI), in honor of the matrix multiplication algorithm from which it emerged, the Method of Four Russians Multiplication (M4RM).

Keywords: Matrix Inversion, Matrix Multiplication, Boolean Matrices, GF(2), Stream Cipher Cryptanalysis, XL Algorithm, Strassen's Algorithm, Method of Four Russians, Gaussian Elimination, LU-factorization.

1 Introduction

Solving a linear system of boolean equations lies at the heart of many cryptanalytic techniques. Examples include stream cipher cryptanalysis via the XL algorithm and its many variants [Arm02, Arm04, AA05, AK03, Cou02, Cou03, Cou04a, CM03, HR04, SPCK00]; the algebraic attacks on the HFE public-key cryptosystem [Cou01, FJ03, CGP03, Cou04b]; and integer factorization methods [AGLL94, PS92], where much of the research on boolean matrices began.

Gaussian Elimination, known in China for at least 2100 years[1] but only known in the West since the 19th century [wiki], is a natural choice of algorithm for these problems. However, for dense systems, its cubic-time complexity makes it far too slow in practice. The algorithm in this paper achieves a speed-up of 3.36 times for a 32000×32000 boolean matrix generated by random fair coins. The theoretical complexity of the algorithm is $O(n^3/\log n)$, but it should be remembered that frequently n is the cube or higher power of a parameter of the system being attacked, and so frequently is in the millions (See Section 6). We also show that this algorithm will out-perform Strassen's algorithm and other "Fast Matrix Multiplication" algorithms developed after Strassen's famous paper, for any reasonably sized matrix. (See Section 1.1 and Appendix A).

[*]Dept. of Applied Mathematics and Scientific Computation

[1]The important Chinese mathematical text *The Nine Chapters on the Mathematical Art*, written about 150 BC, known as *Jiuzhang Suanshu*, has the earliest known presentation of Gaussian Elimination [wiki, KCA].

Origins and Previous Work A paper published by Arlazarov, Dinic, Kronrod, and Faradzev [ADKF70] in 1970 on graph theory contained an $O(n^3/\log n)$ algorithm for finding the transitive closure of a graph. This problem is of course equivalent to exponentiation of a boolean matrix (the adjacency matrix) and the community quickly realized that it was useful not only as a matrix squaring algorithm, but also a matrix multiplication algorithm, because

$$\begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix}^2 = \begin{bmatrix} A^2 & AB \\ 0 & 0 \end{bmatrix}$$

and therefore squaring a matrix and matrix multiplication are equivalent. This equivalency is not as inefficient as it might seem as one can trivially calculate the upper-right quadrant of the answer matrix without calculating the other three-fourths of it. The matrix multiplication algorithm is given in Section 2. This algorithm appears in Aho, Hopcroft, and Ullman's book [AHU], which gives the name "the Method of Four Russians. . . after the cardinality and the nationality of its inventors" [AHU]. While that text states this algorithm is for boolean matrices, one can trivially see how to adapt it to $GF(q)$ for very small q.

A similarly inspired matrix inversion algorithm was known anecdotaly among some cryptanalysts. The author would like to express gratitude to Nicholas Courtois who explained the following algorithm to him after Eurocrypt 2005 in Århus, Denmark. It appears that this algorithm has not been published, either in the literature or on the Internet. The author calls this newer algorithm "Method of 4 Russians Inversion" (M4RI) and the original as "Method of 4 Russians Multiplication" (M4RM).

Contributions Besides giving the details of M4RI, this paper performs the first rigorous analysis of the M4RI algorithm (known to the authors), and also contains experimental verification. In addition, the author maintains coefficients in complexity formulae, while still dropping lower order terms, by using $\sim$ rather than big-Oh notation. The coefficients enable the calculation of cross-over points between two competing algorithms. Moreover, the author proves that Strassen's algorithm cannot be used without modification for solving boolean linear systems, and shows that it is slower than M4RI for any practical sized matrix. Finally, exact (rather than asymptotic) upper-bounds of the running times of some recently published cryptanalytic attacks in the stream cipher, HFE public-key, and integer factorization communities are given. It should be noted that the exact running times can be useful in verifying if an algebraic attack is slower or faster than brute force exhaustive search.

It should be noted that this paper is only concerned with dense systems. Some systems in practice are sparse, and would run much faster under specialized sparse algorithms, such as Lanczos, Wiedemann or Coppersmith's "Block Wiedemann". On the other hand, sparse systems eventually reduce to smaller dense systems and M4RI can be used in these situations as well. Also, one can think of the times given in Section 6 as safe upper-bounds. While they ignore sparsity, no sparse algorithm on a sparse matrix will run slower than the dense approach on that same matrix.

1.1 The Exponent of Matrix Multiplication

A great deal of research was done in the period 1969–1987 on fast matrix operations [Higham, Pan]. Various proofs showed that many important matrix operations, such as QR-decomposition, LU-factorization, inversion, finding determinants, and finding the characteristic polynomial are no more complex than matrix multiplication, in the big-Oh sense [AHU, CLRS]. For this reason, much interest was generated as fast matrix multiplication algorithms were developed [Str69, Sch81, Str87, Pan]. While M4RM and M4RI are specifically for $GF(2)$ (and can work for $GF(q)$ for other very small q with trivial modification) the other algorithms were designed to work over any field by using only the axioms of the definition of a field.

Of the general purpose algorithms, the most famous and frequently implemented of these is Volker Strassen's 1969 algorithm for matrix multiplication. Both Strassen's Algorithm $O(n^{2.81})$ [Str69], and Coppersmith's Algorithm $O(n^{2.38})$ [CW90], the current fastest exponent, have been proposed for boolean linear system solution. Neither, it turns out, is feasible.

Very Low Exponents The algorithms with exponents below $O(n^{2.81})$ all follow the following argument. Matrix multiplication of any particular fixed dimensions is a bilinear map from one vector space to another. The input space is of matrices $\oplus$ matrices as a direct sum, and the output space is another matrix space. Therefore, the map can be written as a tensor. By finding a shortcut for a particular matrix multiplication operation of fixed dimensions, one lower-bounds the complexity[2] of this tensor for those fixed dimensions. Specifically, Strassen performs 2×2 by 2×2 in seven steps instead of eight [Str69]. Likewise, Victor Pan's algorithm performs 70×70 by 70×70 in 143,640 steps rather than 343,000, for an exponent of 2.795 [Pan].

One can now lower-bound the complexity of matrix multiplication in general by extending the shortcut. The method of extension varies by paper, but usually the cross-over[3] can be calculated explicitly. While the actual crossover in practice might vary slightly, these matrices have millions of rows, and are totally infeasible. For example, for Schönhage's algorithm at $O(n^{2.70})$, the crossover is given by [Sch81] at $n = 3^{14} \approx 4.78 \times 10^6$ rows, or $3^{28} \approx 22.88 \times 10^{12}$ entries (this is compared to naïve dense Gaussian Elimination. The crossover would be much higher versus the algorithm in this paper). Note that experimental confirmation of the minimum crossover is impossible, unless one was certain that one had an "optimal implementation," which is a bold claim.

Strassen's Algorithm Strassen's famous paper [Str69] has three algorithms—one for matrix multiplication, one for inversion, and one for the calculation of determinants. The last two are for use with any matrix multiplication algorithm taken as a black-box, and run in time big-Oh of matrix multiplication. However, substantial modification is needed to make these work over GF(2). In the Appendix A, we describe why these modifications are required, and show that Strassen's Formula for Matrix Inversion (used with Strassen's Algorithm for Matrix Multiplication) is slower than M4RI for any feasibly sized matrix. The two algorithms are of equal speed when $n = 6.65 \times 10^6$ rows or 4.4×10^{13} entries.

1.2 The Computational Cost Analysis Model

In papers on matrix operations over the real or complex numbers, the number of floating point operations is used as a measure of running time. This removes the need to account for assembly language instructions needed to manipulate index pointers, iteration counters, features of the instruction set, and measurements of how cache coherency or branch predictions will impact running

[2]An element of a tensor space is a sum of simple tensors. Here, the complexity of a tensor is the smallest number of simple tensors required. This is often called the rank of the tensor, but other authors use the word rank differently. The rank of the tensor is directly proportional to the complexity of the operation [Str87].

[3]The cross-over point is the size where the new tensor has rank (complexity) equal to the naive algorithm's tensor

time. In the present paper, floating point operation counts are meaningless, for boolean matrices do not use floating point operations. Therefore, we propose that matrix entry reads and writes be tabulated, because addition (XOR) and multiplication (AND) are single instructions, while reads and writes on rectangular arrays are much more expensive. Clearly these data structures are non-trivial in size (hundreds of megabytes at the least) and so memory transactions will be the bulk of the computational burden.

From a computer architecture viewpoint in particular, these matrices cannot fit in the cache of the microprocessor, and so the fetches to main memory are a bottleneck. Even if exceptionally careful use of temporal and spatial locality guarantees effective caching (and it is not clear that this is possible), the data must still travel from memory to the processor. The bandwidth of buses has not increased proportionally to the rapid increase in the speeds of microprocessors. Given the relatively simple calculations done once the data is in the microprocessor's registers (i.e. single instructions), it is clear that the memory transactions are the rate-determining step.

Naturally, the exponents of the complexity polynomials of the algorithms in this model will match those of the more flexible big-Oh notation counting all operations. On the other hand, counting exactly the number of memory reads and writes permits us to calculate more directly the coefficients of those polynomials. These coefficients are helpful in three principal ways. First and foremost, they help determine the crossover point at which two competing algorithms are of equal efficiency. This is absolutely critical because matrix multiplication and inversion techniques are often recursive. Second, they help one compare two algorithms of equal exponent. Third, they help calculate the total CPU running time required to solve a system. These coefficients are derived mathematically and are not dependent on machine architectures or benchmarks, but experiments will be performed to confirm them nonetheless. Table 1 has the results of analysis of several matrix operations under this model, taken from the author's E-print [Bar06].

When attempting to convert these memory operation counts into CPU cycles, remember that other instructions are needed to maintain loops, execute field operations, and so forth. Also, memory transactions are not one cycle each, but require several instructions to compute the memory address offsets. Yet, they can be pipe-lined. Thus one can estimate that about 10–30 CPU cycles are needed per matrix memory operation. (Our experiments show about 18, see Section 6).

Matrix Memory Operations Per Second By running the M4RI and dense Gaussian Elimination algorithms on my PC (running 2 GHz, 1 GB RAM, Fedora Linux 5) using `gcc` with all optimizations on, the experiments show running times that suggest 1.11×10^8 matrix memory operations per second is the correct coefficient.

1.3 Notational Conventions

Precise performance estimates are useful, and so rather than the usual five symbols $O(n)$, $o(n)$, $\Omega(n)$, $\omega(n)$, $\Theta(n)$, we will use $f(n) \sim g(n)$ to indicate that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

While $O(n)$ statements are perfectly adequate for many applications, coefficients must be known to determine if algorithms can be run in a reasonable amount of time on particular target ciphers. Also, these coefficients have an enormous impact upon crossover points, as will be shown in the Appendix, comparing Strassen's formula for matrix inversion with the M4RI.

Let $f(n) \leq\sim g(n)$ signify that there exists an $h(n)$ and n_0 such that $f(n) \leq h(n)$ for all $n > n_0$, and $h(n) \sim g(n)$. Equivalently, this means $\limsup f(n)/g(n) \leq 1$ as $n \to \infty$. Matrices in this paper are over GF(2) unless otherwise stated, and are of size m rows and n columns. Denote ℓ as the lesser of n and m. If $n > m$ or $\ell = m$ the matrix is said to be **underdefined**, and if $m > n$ or $\ell = n$ then the matrix is said to be **overdefined**. The symbol $\ln x$ means $\log_e x$ and $\log x$ means $\log_2 x$. The term "n matrix reads/writes" means n operations, all of which are either reads or writes, not n of each.

The term **near-cubic operations** refers to operations of complexity $O(n^3), O(n^3/\log n),$ or $, O(n^{\log_2 7})$. The term **unit upper triangular** means that the matrix is upper triangular, and has only ones on the main diagonal. Obviously such matrices have determinant one, and so are non-singular. Likewise for **unit lower triangular**. A matrix algorithm which seeks to put a matrix into Reduced Row Echelon Form is said to be in RREF mode, and into Unit Upper Triangular Form in UUTF mode. Not all algorithms can do both.

Finally please note that some authors use the term boolean matrix to refer to matrices operating in the semiring where addition is logical OR (instead of logical XOR) and multiplication is AND. Those matrices are used in graph theory, and some natural language processing. Algorithms from the real numbers can work well over this semiring [Fur70], and so are not discussed here.

1.4 To Invert or To Solve?

Generally, four basic options exist when presented with a system of equations, $A\vec{x} = \vec{b}$, over the reals as defined by a square matrix A. First, the matrix A can be inverted, but this is the most computationally intensive option. Second, the system can be adjoined by the vector of constants $\vec{b}$, and the matrix reduced into a triangular form $U\vec{x} = \vec{b'}$ so that the unknowns can be found via back-substitution. Third, the matrix can be factored into LU-triangular form ($A = LUP$), or other forms. Fourth, the matrix can be operated upon by iterative methods to converge to a matrix near to its inverse. Unfortunately, in finite fields concepts like convergence toward an inverse do not have meaning. This rules out option four.

The second option can be extended to solving the same system for two sets of constants $\vec{b_1}, \vec{b_2}$ but requires twice as much work during back substitution. However, back substitution is very cheap (quadratic) compared to the other steps (near cubic). In light of this, it is easy to see that even if many $\vec{b}$ were available, but fewer than $\ell = \min(m, n)$, one would pursue the second option rather than invert the matrix.

Therefore, one would generally choose to put the matrix in unit upper triangular form and use back substitution. Option three, to use LU factorization or another type of factorization will turn out to be the only way to use Strassen's matrix inversion formula. Option one, inverting the matrix would not normally be used. The exception is if the computation is taking place long before the interception of the message, and further the attacker desires the plaintext as soon as possible upon receipt. Then the inverse of the matrix can be calculated, and stored (near cubic cost). Upon receipt of the message, the matrix-times-vector multiplication can take place (quadratic cost).

2 The Four Russians Matrix Multiplication Algorithm

This matrix multiplication algorithm is derivable from the original algorithm published by Arlazarov, Dinic, Kronrod, and Faradzev [ADKF70], and has appeared in books including [AHU].

Consider a product of two matrices $AB = C$ where A is an $a \times b$ matrix and B is a $b \times c$ matrix, yielding an $a \times c$ for C. In this case, one could divide A into b/k vertical "stripes" $A_1 \ldots A_{b/k}$ of k columns each, and B into b/k horizontal stripes $B_1 \ldots B_{b/k}$ of k rows each. (For simplicity assume k divides b). The product of two stripes, $A_i B_i$ is an $a \times b/k$ by $b/k \times c$ matrix multiplication, and yields an $a \times c$ matrix C_i. The sum of all k of these C_i equals C.

$$C = AB = \sum_{i=0}^{i=k} A_i B_i$$

The algorithm itself proceeds as follows:

```
• for $i = 1, 2, \ldots, b/k$ do
    – Make a Gray Code[4] table of all the $2^k$ linear combinations of the $k$ rows of $B_i$.

      Call the $x$th row $T_x$.

      (Costs $(3 \cdot 2^k - 4)c$ reads/writes, see Stage 2, in Section 5).
    – for $j = 1, 2, \ldots, a$ do
        * Read the entries $a_{j,(i-1)k+1}, a_{j,(i-1)k+2}, \ldots, a_{j,(i-1)k+k}$.
        * Let $x$ be the $k$ bit binary number formed by the concatenation of $a_{j,(i-1)k+1}, \ldots, a_{j,ik}$.
        * for $h = 1, 2, \ldots, c$ do
            · Calculate $C_{jh} = C_{jh} + T_{xh}$ (Costs 3 reads/writes).
```

Gray Code Step The step involving the Gray Code requires further explanation. The first entry is all zeroes, and is associated with an all-zero row of appropriate length. The next entry is $0 \cdots 01$, and is associated with the entries from the ith row, from column $i + k$ until column n. The next entry after that is $0 \cdots 011$, and is associated with the entries from the ith and $(i+1)$th rows added together. Since any Gray Code has each line differing from the previous by exactly one bit, only one vector addition is needed to calculate each line of this table. Likewise, with "only" 2^k vector additions, all possible linear combinations of the k rows in the slice have been calculated.

The innermost loop requires 3 steps, the next loop out requires $k + 3c$ steps, and then the next after that requires $(3 \cdot 2^k - 4)c + a(k + 3c)$ steps. Finally the entire algorithm requires

$$\frac{b((3 \cdot 2^k - 4)c + a(k + 3c))}{k} = \frac{3b2^k c - 4cb + abk + 3abc}{k}$$

matrix memory operations. Substitute $k = \log b$, so that $2^k = b$, and observe,

$$\frac{3b^2 c - 4cb + ab \log b + 3abc}{\log b} \sim \frac{3b^2 c + 3abc}{\log b} + ab$$

For square matrices this becomes $\sim (6n^3)/(\log n)$. More on the choice of k can be found in Section 4.

[4] A k-bit Gray Code is all 2^k binary strings of length k, ordered so that each differs by exactly one bit in one position from each of its neighbors. For example, one 3-bit Gray Code is $\{000, 001, 011, 010, 110, 111, 101, 100\}$ [Gra53].

Transposing the Matrix Product Since $AB = C$ implies that $B^T A^T = C^T$, one can transpose A and B, and transpose the product afterward. The transpose is a quadratic, and therefore cheap, operation. This would have running time $(3b^2 a + 3abc)/(\log b) + cb$ and some manipulations will show that this more efficient when $c < a$, for any $b > 1$. Therefore the final complexity is $\sim (3b^2 \min(a, c) + 3abc)/(\log b) + b \max(a, c)$. To see that the last term is not optional, substitute $c = 1$, in which case the last term becomes the dominant term.

Improvements In the years since initial publication, several improvements have been made. In particular, in reducing the memory requirements [AS88, SU86], and the base fields upon which the algorithm can work [San79].

3 The Four Russians Matrix Inversion Algorithm

While the title of this section contains the words "matrix inversion", the algorithm which follows can be used either for matrix inversion, or for triangulation and back-substitution. This is also true for Gaussian Elimination. As stated earlier, even if one has several $\vec{b_1}, \vec{b_2}, \vec{b_3}, \ldots, \vec{b_n}$, it is far more efficient to solve $A\vec{x_i} = \vec{b_i}$ by appending the b_i as columns to the end of matrix A, and putting matrix A in unit upper triangular form (UUTF). Then, one can solve for each x_i by back substitution to obtain the x_i. (This is a quadratic, thus cheap, step). The alternative is to invert A, and Section 3.1 contains changes for that approach, by adjoining A with an identity matrix and processing it into row reduced echelon form (RREF).

In Gaussian Elimination to UUTF of an $m \times n$ matrix, each iteration i operates on the submatrix $a_{ii} \ldots a_{mn}$, with the objective of placing a one at a_{ii} and a zero at every other entry of the column i below row i. In the Method of Four-Russians Inversion (M4RI) algorithm, k columns are processed at once, producing a $k \times k$ identity matrix in the correct spot $(a_{ii} \ldots a_{(i+k-1),(i+k-1)})$, with all zeros below it, and leaving the region above the submatrix untouched.

Stage 1: Denote the first column to be processed in a given iteration as a_i. Then, perform Gaussian elimination on the first $3k$ rows after and including the ith row to produce an identity matrix in $a_{i,i} \ldots a_{(i+k-1),(i+k-1)}$, and zeroes in $a_{(i+k),i} \ldots a_{(i+3k-1),(i+k-1)}$ (To know why it is reasonable to expect this to succeed, see Lemma 1 in Section 5.1).

Stage 2: Construct a table consisting of the 2^k binary strings of length k in a Gray Code. Thus with only 2^k vector additions, all possible linear combinations of these k rows have been precomputed. (See "Gray Code Step" in Section 2).

Stage 3: One can rapidly process the remaining rows from $i + 3k$ until row m (the last row) by using the table. For example, suppose the jth row has entries $a_{ji} \ldots a_{j,(i+k-1)}$ in the columns being processed. Selecting the row of the table associated with this k-bit string, and adding it to row j will force the k columns to zero, and adjust the remaining columns from $i + k$ to n in the appropriate way, as if Gaussian Elimination had been performed.

The process is then repeated $\min(m, n)/k$ times. As each iteration resolves k columns, instead of one column, one could expect that this algorithm is k times faster. The trade-off for large k is that Stage 2 can be very expensive. It turns out (see Section 4) that selecting the right value of k is critical.

3.1 Triangulation or Inversion?

While the above form of the algorithm will reduce a system of boolean linear equations to unit upper triangular form, and thus permit a system to be solved with back substitution, the M4RI algorithm can also be used to invert a matrix, or put the system into reduced row echelon form (RREF). Simply run Stage 3 on rows $0 \cdots i-1$ as well as on rows $i+3k \cdots m$. This only affects the complexity slightly, changing the 2.5 coefficient to 3 (details omitted, see the author's E-print [Bar06]). To use RREF to invert a matrix, simply concatenate an identity matrix (of size $n \times n$) to right of the original matrix (of size $n \times n$), producing a $n \times 2n$ matrix. Using M4RI to reduce the matrix to RREF will result in an $n \times n$ identity matrix appearing on the left, and the inverse matrix on the right.

4 Experimental and Numerical Results

Five experiments were performed. The first was to determine the correct value of k for M4RI. The second was to determine the running time of both M4RI and Gaussian Elimination. In doing these experiments, the author noted that the optimization level of the compiler heavily influenced the output. Therefore, the third experiment attempted to calculate the magnitude of this influence. The fourth was to determine if a fixed k or flexible k was superior for performance.

The specifications of the computer on which the experiments were run is given at the end of Section 1.2. Except as noted, all were compiled under `gcc` with the highest optimization setting (level three). The experiments consisted of generating a matrix filled with fair coins, and then checking the matrix for invertibility by attempting to calculate the inverse using M4RI to RREF. If the matrix was singular, a new matrix was generated. If the matrix was invertible, the inverse was calculated again using Gaussian Elimination to RREF. These two inverses were then checked for equality, and finally one was multiplied by the original to obtain a product matrix which was compared with the identity matrix. The times were calculated using `clock()` from `time.h` built into the basic `C` language. The functions were all timed independently, so extraneous operations like verifying the correctness of the inverse will not affect running time (except possibly via cache coherency but this is both unlikely and hard to detect). No other major tasks were being run on the machine during the experiments, but `clock()` measures user-time and not time in the sense of a wall clock.

In the first experiment (to determine the best value of k), the range of k was permitted to change. The specific k which resulted in the lowest running time was reported for 30 matrices. Except when two values of k were tied for fastest (recall that `clock()` on Linux has a granularity of 0.01 sec), the thirty matrices were unanimous in their preferred value of k in all cases. A linear regression shows that $k = c_1(\log n) + c_2$ has minimum error in the mean-squared sense at $k = (3/4)(\log n) + 0$. For the next two experiments, k was fixed to be eight to simplify addressing. Another observed feature of the first experiment was that the running time was trivially perturbed if the value of k was off by one, and by a few percent if off by two. The results are in Table 3.

Each trial of the second experiment consisted of the same code compiled under all four optimization settings. Since k was fixed to eight, addressing was vastly simplified and so the program was rewritten to take advantage of this. The third experiment simply used the code from the second experiment, with the compilation set to optimization level 3. The results are in Table 6 and Table 5. The fourth experiment had the best running times, because k was permitted to vary. This was a surprise, because the addressing difficulties were nontrivial, and varying k slightly has a small effect on running time. See Table 4 for the affect of relatively adjusting k upward or downward.

A fifth mini-experiment was to take the computational cost expression for M4RI, and place it into a spreadsheet, to seek optimal values of k for very large values of n, for which experimentation would not be feasible. The expression $1 + \log n - \log\log n$ was a better fit than any $c_1 + c_2 \log n$. On the other hand, it would be very hard to determine the coefficient of the $\log\log n$ term in that expression, since a double logarithm differs only slightly from a line.

5 Exact Analysis of Complexity

Assume for simplicity that $\log n$ divides n and m. To calculate the cost of the algorithm one need only tabulate the cost of each of the three stages, which will be repeated $\min(m,n)/k$ times. Let these stages be numbered $i = 1 \ldots \min(m,n)/k$.

The first stage is a $3k \times n - ik$ underdefined Gaussian Elimination (RREF), which requires $\sim 1.5(3k)(n-ik)^2 - 0.75((3k)^3)$ matrix memory operations (see the author's E-print [Bar06]). This will be negligible.

The second stage, constructing the table, requires $3(n-ik-k)$ steps per row. The first row is all zeroes and can be hard-coded, and the second row is a copy of the appropriate row of the matrix, and requires $(n-ik-k)$ reads followed by writes. Thus one obtains $2(n-ik-k)+(2^k-2)(3)(n-ik-k) = (3 \cdot 2^k - 4)(n - ik - k)$ steps.

The third stage, executed upon $(m - ik - 3k)$ rows (if positive) requires $2k + 3(n - ik - k)$ reads/writes per row. This becomes $(m-ik-3k)(3n-3ik-k)$ matrix memory operations in total, when that total is positive. For example, in a square matrix the last 2 iterations of stage 1 will take care of all of these rows and so there may be no work to perform in Stage 3 of those iterations. To denote this, let $pos(x) = x$ if $x > 0$ and $pos(x) = 0$ otherwise.

Adding steps one, two and three yields

$$\begin{aligned}
& \sum_{i=0}^{i=\ell/k-1} 1.5(3k)^2(n-ik) - 0.75((3k)^3)(3\cdot 2^k - 4)(n-ik-k) + (pos(m-ik-3k))(3n-3ik-k) \\
= & \left[\sum_{i=0}^{i=\ell/k-3} 1.5(3k)^2(n-ik) - 0.75((3k)^3)(3\cdot 2^k - 4)(n-ik-k) + (m-ik-3k)(3n-3ik-k)\right] \\
& +1.5(3k)^2(n-\ell+2k) - 0.75((3k)^3)(3\cdot 2^k - 4)(n-\ell+k) \\
& +1.5(3k)^2(n-\ell+k) - 0.75((3k)^3)(3\cdot 2^k - 4)(n-\ell) \\
\leq & \sim \frac{1}{4k}\left[2^k(-6k\ell + 12n\ell - 6\ell^2) - 6m\ell^2 - 6n\ell^2 + 4\ell^3 + 12mn\ell\right]
\end{aligned}$$

Recalling $\ell = \min(m,n)$ and substituting $k = \log \ell$ and thus $2^k = \ell$,

$$\frac{1}{4\log\ell}\left(6n\ell^2 - 2\ell^3 - 6m\ell^2 + 12mn\ell\right)$$

Thus for the over-defined case ($\ell = n$) this is $(4n^3 + 6n^2m)/(4\log n)$, and for the under-defined case ($\ell = m$) this is $(18nm^2 - 8m^3)/(4\log m)$, and for square $(5n^3)/(2\log n)$.

5.1 The Rank of $3k$ Rows, or Why $k + \epsilon$ is not Enough

The reader may be curious why $3k$ rows are selected instead of k rows at the small Gaussian Elimination step (Stage 1 of each iteration). Normally to guarantee non-singularity, a system with k variables is solved with $k + \epsilon$ equations, where $\epsilon \approx 2 \ldots 100$. However, this does not work in the M4RI algorithm, because $\ell / \log \ell$ submatrices must be reduced by Gaussian Elimination, and the algorithm fails if any of these submatrices are singular.

The answer is that the probability of k vectors of length $3k$ having rank k is very high, as proved below. The small Gaussian Elimination will fail to produce the identity matrix followed by rows of zeroes if and only if this submatrix is not of full rank.

Lemma 1 *A random boolean matrix of dimension $3k \times k$, filled by fair coins, has full rank with probability $\approx 1 - 2^{-2k}$.*

Proof: Consider the columns of the matrix as vectors. One can attempt to count the number of possible full rank matrices. The first vector can be any one of $2^{3k} - 1$ non-zero length $3k$ vectors. The second one can be any non-zero vector distinct from the first, or $2^{3k} - 2$ choices. The third one can be any non-zero vector not equal to the first, the second, or their sum, or $2^{3k} - 4$. The ith vector can be any vector not in the space spanned by the previous $i - 1$ vectors (which are linearly independent by construction). Thus $2^{3k} - 2^{i-1}$ choices are available. Therefore, the probability of any k vectors of length $3k$ being linearly independent is

$$\frac{\prod_{i=1}^{i=k} (2^{3k} - 2^{i-1})}{(2^{3k})^k} = \prod_{i=1}^{i=k} (1 - 2^{i-1} 2^{-3k}) \approx 1 - \sum_{i=1}^{i=k} 2^{i-1} 2^{-3k} \approx 1 - 2^{-3k}(2^k - 1) \approx 1 - 2^{-2k}$$

And this is the desired result. []
In the case $k = 5$, the actual probability of less than full rank is 9.46×10^{-4}, and the above formula has a relative error of 3.08×10^{-6}, and would be even more accurate for higher k. Also, note when $k = \log \ell$ then the probability of full rank is $1 - \ell^{-2}$. Since there will be $(\ell)/(\log \ell) - 1$ iterations, the probability of even one failure during all passes is approximately $1/(\ell \log \ell)$, which is very low, considering that ℓ may approach the millions. (See Section 6 for typical sizes).

Note that even if $2k \times k$ were chosen, then the probability of failure over the whole algorithm would be $1/\log \ell$, which is non-trivial. Finally it is interesting to note that approximately 29% of boolean square random matrices are invertible, and 71% are singular. To calculate this, redo the proof of the lemma with k instead of $3k$. In practice, when k was significantly lower than $\log \ell$, the algorithm would abort very frequently, whereas it never aborted in any of the experiments when k was set near $\log \ell$. (Abortions marked with a star in Table 3).

5.2 Using Bulk Logical Operations

The above algorithm can be improved upon if the microprocessor has instructions for 32-bit (or even 64-bit) logical operations. Stages 2 and 3 essentially consist of repeated row additions. The matrix can be stored in an 8-bits per byte format instead of the 1-bit per byte format, and long XOR operations can perform these vector additions. Stage 1 is unaffected. However, stages 2 and 3 can proceed 32 or 64 times as fast as normal if single-instruction logical operators are available in those sizes, as they are on all modern PCs. Since only stages 2 and 3 were non-negligible, it is safe to say that the algorithm would proceed 32 or 64 times faster, for sufficiently large matrices.

Experimentally the author found that the speed up varied between 80% to 95% of this figure, depending on the optimization settings of the compiler chosen. However, there is absolutely no reason not to do this all the time, so the vector additions were performed 64 entries at one time.

6 Applications to Cryptanalysis

It is useful to calculate the cost of solving a system of equations. The final cost of a system of m equations in n unknowns is equivalent to performing M4RI in UUTF mode on a $m \times (n+1)$ matrix. The extra column is for the constants. According to Table 1, this will cost $\sim (n^3+1.5n^2m)/(64\log n)$ for $m < n$ and $\sim (4.5nm^2 - 2m^3)/(64\log m)$ for $m > n$. We neglect the back substitution which costs $\sim \min(m,n)^2 + n + m$. This then should be divided by the previous figure of 1.11×10^8 matrix memory operations per second.

6.1 Stream Ciphers

First, in stream cipher cryptanalysis, a stream cipher is converted to a system of boolean polynomial equations whose solution is the key to the cipher. Second, the polynomial system is then converted to a boolean linear system. This system is often sparse but its density will rise very rapidly under Gaussian Elimination. Finally, the dense linear boolean system must be solved. The number of equations or rows is m, and the number of variables or columns is n.

- Courtois and Meier attack Toyocrypt with $\binom{128}{3} = 341,376$ equations [CM03]. The system is slightly overdefined, so therefore near square.
- Courtois attacks Sfinks $2^{22.5}$ equations in one case and $2^{36.5}$ in another [Cou05]. The system is slightly overdefined, so therefore near square.
- Hawkes and Rose state that the best known attacks against E-0, Lili-128, and Toyocrypt have $2^{18}, 2^{12}$, and 2^7 equations. It is clear from context that the matrices are nearly square [Arm02, HR04].

6.2 Hidden Field Equations

Courtois suggested attacking the HFE Challenge 1 with a dense boolean matrix of dimension 1,831,511 [Cou01]. This is not the fastest attack, see [FJ03].

6.3 Integer Factorization

In integer factorization, once all the relations have been found in an algorithm similar to the Quadratic Field Sieve, one must find a vector in the null space of a matrix. This can be done by first preconditioning the matrix [PS92], then using Sparse Gaussian Elimination and finally a applying dense matrix operation step on a matrix of reduced size [AGLL94].

The sizes of this final dense matrix operation in practice can be described by the process to factor the 129-digit prime number protecting the message "the magic words are squeamish ossifrage" from the paper of the same name [AGLL94]. A sparse $569,466 \times 524,339$ boolean matrix was reduced to a dense $188,614 \times 188,160$ dense boolean matrix, which was solved in 12 CPU hours of a Sun Sparc 10 workstation at that time (see below for today's estimate).

6.4 Algebraic Attacks on Block Ciphers

In Appendix B.5 of Courtois and Pieprzyk [CP02], two attacks on Rijndael are given which involve solving a system of quadratic equations over GF(2). It is unclear how large the matrices for these will be, as this depends on the conversion from a polynomial to a linear system. However, if these attacks ever become feasible, M4RI is a good choice for the final linear algebra step. Likewise, the complexity results here and in [Bar06] would be useful for determining if the attack is slower or faster than brute force search.

6.5 Feasibility Analysis

During the actual testing, the performance times were divided by the values given by the model (e.g. $(9/64)n^3/\log(n)$ for a $n \times 2n$ underdefined matrix, with M4RI in RREF mode), to obtain 1.11×10^8 matrix memory operations per second on the author's PC. (See the end of Section 1.2 for details). Since it is a 1.14 GHz machine, the author suggests that 10^8 matrix memory operations per second per GHz is good at least as an order of magnitude estimate, or first approximation. Alternatively, this is 11.4 clock cycles per matrix memory operation. This comes to 3.50×10^{15} matrix memory operations per year.

To give some scale to the feasibility of the algorithms as applied to the attacks discussed here, note that a Cray X1E[5] is a 174 Tflop[6] machine, performing 4.6×10^{21} floating point operations in a year. It is *not correct* to equate a 11.4 Pentium IV clock cycles per matrix operation with 11.4 flops per matrix operation, since they are not floating point operations, but it will serve as an estimate. This comes to 1.28×10^{13} matrix memory operations per second, or 4.04×10^{20} matrix memory operations per year.

The figures above permit one to calculate how long the above attacks would take, ignoring the effects of sparsity which may be very important in some cases. These are calculated using the figures 4.04×10^{20} or 3.5×10^{15} matrix memory operations per Cray-year or per PC-year from above, and M4RI in UUTF mode.

Attack	Matrix Mem Ops	CRAY Time	PC Time
Courtois and Meier vs. Toyocrypt:	8.45×10^{14}	66 secs	88 days
Courtois vs. Sfinks ($2^{22.5}$ case):	3.62×10^{18}	3.3 days	1030 yrs
Courtois vs. Sfinks ($2^{36.5}$ case):	9.82×10^{30}	2.43×10^{10} yrs	2.80×10^{15} yrs
Best known attack on E-0:	3.9×10^{13}	3.0 secs	4.0 days
Best known attack on Lili-128:	2.23×10^8	trivial	2 secs
Squeamish Ossifrage:	1.49×10^{14}	12 secs	15.5 days
HFE Challenge 1:	4.15×10^{16}	54 mins	11.8 yrs

One can conclude four things from this list. First, that matrix operation running times are highly dependent on dimension. Second, that algebraic attacks are indeed feasible against all but one of the above systems. Third, that some attacks will require some sort of supercomputing facility.

[5]Data from the Cray website.

[6]For comparison the SETI@Home system operates at 250 TFlop [wiki2].

Fourth, systems not feasible in a day are probably not feasible in a year either, since $365^{1/3} \approx 7.14$. However, one should also mention that while showing a quick running time is proof an attack is feasible, showing a slow running time is not proof of infeasibility, because sparse methods may be successful for the billion year dense attack above.

7 Acknowledgments

The author would like to recognize first and foremost, Nicholas Courtois, whose work inspired this paper and whose ideas, so freely given in conversation, have been indispensable. We are deeply indebted to Lawrence Washington and Patrick Studdard for comments on earlier drafts of this paper. The author would also like to recognize Dan Bernstein for his encouragement and detailed suggestions. Antoine Joux was very helpful by sharing an algorithm for calculating the Gray Code on-the-fly. Much of this work was done by the author in Paris, supported by ECRYPT, a European Union "Center of Excellence", as an invited visiting scientist.

References

[AHU] A. Aho, J. Hopcroft, and J. Ullman. "Chapter 6: Matrix Multiplication and Related Operations." *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AGLL94] D. Atkins, M. Graff, A. Lenstra, P. Leyland. "The Magic Words are Squeamish Ossifrage." *Advances in Cryptology: ASIACRYPT'94* 1994.

[ADKF70] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. "On Economical Construction of the Transitive Closure of a Directed Graph." *Dokl. Akad. Nauk. SSSR* No. 194 (in Russian), English Translation in *Soviet Math Dokl.* No. 11, 1970.

[Arm02] F. Armknecht. "A Linearization Attack on the Bluetooth Key Stream Generator." IACR E-print, 2002, No. 191.

[Arm04] F. Armknecht. "Improving Fast Algebraic Attacks." Proceedings of Fast Software Encryption '04, Springer-Verlag *Lecture Notes in Computer Science*, 2004.

[AA05] F. Armknecht, and G. Ars. "Introducing a New Variant of Fast Algebraic Attacks and Minimizing Their Successive Data Complexity." Proceedings of Mycrypt '05, Springer-Verlag *Lecture Notes in Computer Science*, 2005.

[AK03] F. Armknecht and M. Krause. "Algebraic Attacks on Combiners with Memory." *Advances in Cryptology:* CRYPTO 2003.

[AS88] M. Atkinson and N. Santoro. "A Practical Algorithm for Boolean Matrix Multiplication." *Information Processing Letters.* 13 September 1988.

[Bar06] G. Bard. "Achieving a log(n) Speed Up for Boolean Matrix Operations and Calculating the Complexity of the Dense Linear Algebra step of Algebraic Stream Cipher Attacks and of Integer Factorization Methods." IACR E-print, 2006, No. 163.

[Ber95] D. Bernstein. "Matrix Inversion Made Difficult." Unpublished Manuscript available on `http://cr.yp.to/papers/mimd.ps`

[BH74] J. Bunch and J. Hopcroft. "Triangular Factorization and Inversion by Fast Matrix Multiplication." *Math Comp.* No. 28:125, 1974.

[CLRS] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. "Chapter 28: Matrix Operations." *Introduction to Algorithms, Second Edition.* MIT Press, McGraw-Hill Book Company, 2001.

[CW90] D. Coppersmith and S. Winograd. "Matrix Multiplication via Arithmetic Progressions." *J. of Symbolic Computation*, 9. 1990.

[Cou01] N. Courtois. "The security of Hidden Field Equations (HFE)", Cryptographers' Track Rsa Conference, 2001.

[Cou02] N. Courtois. "Higher Order Correlation Attacks, XL Algorithm and Cryptanalysis of Toyocrypt." Proceedings of ICISC '02, Springer-Verlag *Lecture Notes in Computer Science*, 2002.

[Cou03] N. Courtois. "Fast Algebraic Attacks on Stream Ciphers with Linear Feedback." *Advances in Cryptology:* CRYPTO 2003.

[Cou04a] N. Courtois. "Algebraic Attacks on Combiners with Memory and Several Outputs." Proceedings of ICISC '04, Springer-Verlag *Lecture Notes in Computer Science*, 2004.

[Cou04b] N. Courtois. "Short Signatures, Provable Security, Generic Attacks and Computational Security of Multivariate Polynomial Schemes such as HFE, Quartz and Sflash." IACR E-print, 2004, No. 143.

[Cou05] N. Courtois. "Cryptanalysis of Sfinks." Proceedings of ICISC '05, Springer-Verlag *Lecture Notes in Computer Science*, 2005.

[CGP03] N. Courtois, L. Goubin, and J. Patarin. "SFLASHv3, a fast asymmetric signature scheme." IACR E-print, 2003, No. 211.

[CM03] N. Courtois, and W. Meier. "Algebraic Attacks on Stream Ciphers with Linear Feedback." *Advances in Cryptology:* EUROCRYPT 2003.

[CP02] N. Courtois, and J. Pieprzyk. "Cryptanalysis of Block Ciphers with Overdefined Systems of Equations." *Advances in Cryptology:* ASIACRYPT 2002.

[FJ03] J. Faugère, and A. Joux. "Algebraic Cryptanalysis of Hidden Field Equation (HFE) Cryptosystems Using Gröbner Bases" *Advances in Cryptology:* CRYPTO 2003.

[Fur70] M. Furman. "Application of a Method of Fast Multiplication of Matrices in the Problem of Finding the Transitive Closure of a Graph." *Dokl. Akad. Nauk. SSSR* No. 194 (in Russian), English Translation in *Soviet Math Dokl.* No. 3, 1970.

[Gra53] F. Gray. *Pulse Code Communication.* March 17, 1953. USA Patent 2,632,058.

[HR04] P. Hawkes, and G. Rose. "Rewriting Variables: the Complexity of Fast Algebraic Attacks on Stream Ciphers." *Advances in Cryptology:* CRYPTO 2004.

[Higham] N. Higham. "Chapter 23: Fast Matrix Multiplication." *Accuracy and Stability of Numerical Algorithms, Second Edition.* SIAM, 2002.

[KCA] S. Kangshen, J. Crossley, and L. Anthony (Eds.) *The Nine Chapters on the Mathematical Art.* Oxford University Press. 1999.

[Moo20] E. Moore. "On the Reciprocal of the General Algebraic Matrix." *Bulletin of the American Mathematical Society.* 26, 1920.

[Pan] V. Pan. "Chapter 1: The Exponent of Matrix Multiplication." *How to Multiply Matrices Faster.* Springer-Verlag, 1984.

[Pen55] R. Penrose "A Generalized Inverse for Matrices." *Proc. of the Cambridge Phil. Soc.* 51. 1955.

[PS92] C. Pomerance and J. Smith. "Reduction of Huge, Sparse Matrices over Finite Fields via Created Catastrophes." *Experimental Mathematics.* Vol. 1, No. 2. 1992.

[San79] N. Santoro. "Extending the Four Russians' Bound to General Matrix Multiplication." *Information Processing Letters.* 18 March 1979.

[SU86] N. Santoro and J. Urrutia. "An Improved Algorithm for Boolean Matrix Multiplication." *Computing*, 36. 1986.

[Sch81] A. Schönhage. "Partial and Total Matrix Multiplication." SIAM Journal of Computing, Vol 10, No. 3, August 1981.

[SPCK00] A. Shamir, J. Patarin, N. Courtois, and A. Klimov. "Efficient Algorithms for solving Overdefined Systems of Multivariate Polynomial Equations." *Advances in Cryptology:* CRYPTO 2004.

[Str69] V. Strassen. "Gaussian Elimination is not Optimal." Numerische Mathematik, Vol 13, No 3. 1969.

[Str87] V. Strassen. "Relative Bilinear Complexity and Matrix Multiplication." J. Reine Angew. Math. Vols 375 & 376, 1987.

[wiki] "Gaussian Elimination." From Wikipedia—The Free Encyclopedia.

[wiki2] "SETI@Home." From Wikipedia—The Free Encyclopedia.

A The Unsuitability of Strassen's Algorithm

It is important to note that Strassen's famous paper [Str69] has three algorithms. The first is a matrix multiplication algorithm, which we call "Strassen's Algorithm for Matrix Multiplication." The second is a method for using any matrix multiplication technique for matrix inversion, in asymptotically equal time (in the big-Oh sense). We call this Strassen's Formula for Matrix Inversion. The third is a method for the calculation of the determinant of a matrix. Below, Strassen's Formula for Matrix Inversion is analyzed, by which a system of equations over a field can be solved.

Given a square matrix A, by dividing it into equal quadrants one obtains the following inverse: (A more detailed exposition is found in [CLRS], using the same notation).

$$A = \begin{bmatrix} B & C \\ D & E \end{bmatrix} \quad \Rightarrow \quad A^{-1} = \begin{bmatrix} B^{-1} + B^{-1}CS^{-1}DB^{-1} & -B^{-1}CS^{-1} \\ -S^{-1}DB^{-1} & S^{-1} \end{bmatrix}$$

where $S = E - DB^{-1}C$, which is the Schur Complement of A with respect to B.

One can easily check that the product of A and the matrix formula for A^{-1} yields the identity matrix, either multiplying on the left or on the right. If an inverse for a matrix exists, it is unique, and so therefore this formula gives the unique inverse of A, provided that A is in fact invertible.

However, it is a clear requirement of this formula that B and S be invertible. Over the real numbers, or other subfields of the complex numbers, one can show that if A and B are non-singular, then S is non-singular also [CLRS]. The problem is to guarantee that the upper-left submatrix, B, is invertible. Strassen did not address this in the original paper, but the usual solution is as follows (more details found in [CLRS]). First, if A is positive symmetric definite, then all of its principal submatrices are positive symmetric definite, including B. All positive symmetric definite matrices are non-singular, so B is invertible. Now, if A is not positive symmetric definite, but is non-singular, then note that A^TA is positive symmetric definite and that $(A^TA)^{-1}A^T = A^{-1}$. This also can be used to make a pseudoinverse for non-square matrices, called the Moore-Penrose Pseudoinverse [Moo20, Pen55, Ber95].

However, the concept of positive symmetric definite does not work over a finite field, because these fields cannot be ordered (in the sense of an ordering that respects the addition and multiplication operations). Observe the following counterexample,

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \qquad A^TA = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Both A and A^TA have det $= 1$, thus are invertible. Yet in both cases the upper-left hand 2x2 submatrices have det $= 0$, and therefore are not invertible. Thus Strassen's formula for inversion is unusable without modification. The modification below is from Aho, Hopcroft and Ullman's book [AHU] though first appeared in [BH74].

Consider a matrix L that is unit lower triangular, and a matrix U that is unit upper triangular. Then Strassen's Matrix Inversion Formula indicates

$$L = \begin{bmatrix} B & 0 \\ D & E \end{bmatrix} \quad \Rightarrow \quad L^{-1} = \begin{bmatrix} B^{-1} & 0 \\ -E^{-1}DB^{-1} & E^{-1} \end{bmatrix}$$

$$U = \begin{bmatrix} B & C \\ 0 & E \end{bmatrix} \quad \Rightarrow \quad U^{-1} = \begin{bmatrix} B^{-1} & -B^{-1}CE^{-1} \\ 0 & E^{-1} \end{bmatrix}$$

Note $S = E - DB^{-1}C$, which is the Schur Complement of A with respect to B, becomes $S = E$ in both cases, since either C or D is the zero matrix. Since L (or U) is unit lower (or upper) triangular, then its submatrices B and E are also unit lower (or upper triangular), and therefore invertible. Therefore Strassen's Matrix Inversion Formula over GF(2) will always work for unit lower or upper triangular matrices.

It is well known that any matrix over any field has a factorization $A = LUP$ where P is a permutation matrix, L is unit lower triangular and U is unit upper triangular [CLRS]. Once A is thus factored, the matrix inversion formula is sufficient to calculate A^{-1}. Aho, Hopcroft and Ullman [AHU] give an algorithm for computing the LUP factorization over an arbitrary field, in time equal to big-Oh of matrix multiplication, by use of a black-box matrix multiplication algorithm. We call this algorithm AHU-LUP. The algorithm is very complex and is not given here, but a detailed analysis is found in the author's E-print [Bar06]. Once the factorization of A is complete, Strassen's Matrix Inversion Formula can be applied to L and U. Note $P = P^{-1}$, and also $A^{-1} = P^{-1}U^{-1}L^{-1}$.

Consequences Therefore to use the Strassen Matrix Inversion Formula, we calculate the LUP-factorization, invert both L and U, and finally perform matrix multiplication to get A^{-1}. These extra steps, which are each of equal complexity to the whole algorithm in the big-Oh sense, add a large hidden coefficient to $O(n^{2.81})$, and make the crossover with Gaussian Elimination enormous.

This coefficient is calculated explicitly in the author's E-print [Bar06], and is $\sim 6.4893n^{2.81}$ matrix memory operations, for inverting a boolean matrix. The equivalent value for M4RI is $\sim 9n^3/(64 \log n)$ (this is an underdefined $n \times 2n$ matrix, processed with M4RI into RREF). Simple calculation shows that these two expressions are equal for $n = 1.33 \times 10^{18}$ rows. Recall, this is the point where the two algorithms are of equal performance. The point at which Strassen is twice as fast as the M4RI is $n = 8.44 \times 10^{20}$ rows. These matrices are have 1.78×10^{36} and 7.11×10^{39} entries respectively. For this reason, Strassen's Formula for Matrix Inversion is not faster than the method of Four Russians for any size matrix whose inverse can be computed at present.

However, if one merely wants to solve a system of equations, one can perform the LU-factorization using AHU-LUP with Strassen's Matrix Multiplication in the black box, and then use back-substitution, without calculating A^{-1} explicitly. This changes the coefficients to $\sim 2.2876n^{2.81}$ and $\sim 2.5n^3/(64 \log n)$. The cross-over is therefore at 5.59×10^{18}, and the doubling point 3.47×10^{20}.

Parralelization Note that Strassen's Algorithm for Matrix Multiplication and Strassen's Matrix Inversion Formula are potentially amenable to parralelization, since no communication between processes is required once A is cut into quadrants [CLRS]. Therefore, in the presence of 4^n processors Strassen's Algorithm or Formula could be useful in breaking the problem up n times into 4^n pieces, each of which can be solved independently via the M4RI or M4RM as required. The complexity, however, would still depend on the underlying algorithm (M4RI or M4RM) and there exist circumstances in which it would be better to run M4RI on one processor and leave all the others idle than it would to run Strassen's algorithms (see the author's E-print [Bar06]).

Finally, note that it might be possible to find a method of speeding up Strassen by a factor of 64, using 64-bit arithmetic. This would make the cross-overs 9.63×10^6 and 1.40×10^6 for system-solving and matrix inversion, respectively.

Table 1: Inversion Algorithms and Asymptotic Performance

Algorithm	Overdefined	Square	Underdefined
System Upper-Triangularization			
M4RI (UUTF)	$(n^3 + 1.5n^2m)/(\log n)$	$(2.5n^3)/(\log n)$	$(4.5nm^2 - 2m^3)/(\log m)$
AHU-LUP Fact. with Strassen	—	$2.2876n^{\log 7}$	$2.2876m^{\log 7} + 3mn$
Dense Gaussian Elim. (UUTF)	$0.75mn^2 - 0.25n^3$	$0.5n^3$	$0.75nm^2 - 0.25m^3$
Back Substitution	$n^2 + m$	n^2	$m^2 + n$
Matrix Inversion			
Strassen's—U/L Triang. Inversion	—	$0.9337n^{\log 7}$	—
Strassen's—General Inversion	—	$6.4893n^{\log 7}$	—
Dense Gaussian Elim. (RREF)	$0.75mn^2$	$0.75n^3$	$1.5nm^2 - 0.75m^3$
M4RI (RREF)	$(3n^3 + 3mn^2)/(2\log n)$	$(3n^3)/(\log n)$	$(6nm^2 - 3m^3)/(\log m)$

Table 2: Multiplication Algorithms and Asymptotic Performance

Algorithm	Rectangular $a \times b$ by $b \times c$	Square $n \times n$ by $n \times n$
Multiplication		
M4RM	$(3b^2 \min(a, c) + 3abc)/(\log b) + b \max(a, c)$	$(6n^3)/(\log n)$
Naïve Multiplication	$2abc$	$2n^3$
Strassen's Algorithm	$2.3343(abc)^{(\log 7)/(3)}$	$2.3343n^{\log 7}$

Table 3: Running times, in msec, Optimization Level 0

k	1,024	1,536	2,048	3,072	4,096	6,144	8,192	12,288	16384
5	870	2,750	6,290	20,510	47,590	—*	—*	—*	–
6	760	2,340	5,420	17,540	40,630	132,950	—*	1,033,420	–
7	710	2,130	4,850	15,480	35,540	116,300	—*	903,200	–
8	680	2,040	4,550	14,320	32,620	104,960	242,990	798,470	–
9	740	2,100	4,550	13,860	30,990	97,830	223,270	737,990	1,703,290
10	880	2,360	4,980	14,330	31,130	95,850	215,080	690,580	1,595,340
11	1,170	2,970	5,940	16,260	34,020	99,980	218,320	680,310	1,528,900
12	1,740	4,170	7,970	20,470	41,020	113,270	238,160	708,640	1,557,020
13	2,750	6,410	11,890	29,210	55,970	147,190	295,120	817,950	1,716,990
14	4,780	10,790	19,390	45,610	84,580	208,300	399,810	1,045,430	–
15	8,390	18,760	33,690	77,460	140,640	335,710	623,450	1,529,740	–
16	15,290	34,340	60,570	137,360	246,010	569,740	1,034,690	2,440,410	–

*Indicates that too many abortions occured due to singular submatrices. See Section 5.1.

Table 4: Percentage Error for Offset of K, From Experiment 1

error of k		1,024	1,536	2,048	4,096	6,144	8,192	12,288	16384
-4	—	—	48.0%	53.6%	38.7%	–	32.8%	—	
-3	27.9%	34.8%	26.6%	31.1%	21.3%	–	17.4%	—	
-2	11.8%	14.7%	11.7%	14.7%	9.5%	13.0%	8.5%	11.4%	
-1	4.4%	4.4%	3.3%	5.3%	2.1%	3.8%	1.5%	4.3%	
Exact	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	
+1	8.8%	2.9%	3.4%	0.5%	4.3%	1.5%	4.2%	1.8%	
+2	29.4%	15.7%	17.3%	9.8%	18.2%	10.7%	20.2%	12.3%	
+3	72.1%	45.6%	47.7%	32.4%	53.6%	37.2%	53.7%	—	
+4	155.9%	104.4%	110.8%	80.6%	117.3%	85.9%	124.9%	—	
+5	304.4%	214.2%	229.1%	172.9%	250.2%	189.9%	258.7%	—	
+6	602.9%	428.9%	458.9%	353.8%	494.4%	381.1%	—	—	

Table 5: Results of Experiment 3—Running Times, Fixed k=8

Size	M4RI	Gaussian	Ratio
4,000 rows	18.97 s	6.77 s	2.802
6,000 rows	59.40 s	22.21 s	2.674
8,000 rows	135.20 s	51.30 s	2.635
12,000 rows	167.28 s	450.24 s	2.692
16,000 rows	398.12 s	1023.99 s	2.572
20,000 rows	763.92 s	1999.34 s	2.617

Table 6: Experiment 2—Running times in seconds under different Optimizations, k=8

	Opt 0	Opt 1	Opt 2	Opt 3
4000 x 4000				
Gauss	91.41	48.35	48.37	18.97
Russian	29.85	17.83	17.72	6.77
Ratio	3.062	2.712	2.730	2.802
6000 x 6000				
Gauss	300.27	159.83	159.74	59.40
Russian	97.02	58.43	58.38	22.21
Ratio	3.095	2.735	2.736	2.674
8000 x 8000				
Gauss	697.20	371.34	371.86	135.20
Russian	225.19	136.76	135.21	51.30
Ratio	3.096	2.715	2.750	2.635

Table 7: Optimization Level 3, Flexible k

Dimension	4,000	8,000	12,000	16,000	20,000	24,000	28,000	32,000
Gaussian	19.00	138.34	444.53	1033.50	2022.29	3459.77	5366.62	8061.90
7	7.64	–	–	–	–	–	–	–
8	7.09	51.78	–	–	–	–	–	–
9	6.90	48.83	159.69	364.74	698.67	1195.78	–	–
10	7.05	47.31	151.65	342.75	651.63	1107.17	1740.58	2635.64
11	7.67	48.08	149.46	332.37	622.86	1051.25	1640.63	2476.58
12	–	52.55	155.51	336.11	620.35	1032.38	1597.98	2397.45
13	–	–	175.47	364.22	655.40	1073.45	1640.45	2432.18
14	–	–	–	–	–	–	1822.93	2657.26
Min	6.90	47.31	149.46	332.37	620.35	1032.38	1597.98	2397.45
Gauss/M4RI	2.75	2.92	2.97	3.11	3.26	3.35	3.36	3.36