Abstract

- Abstract. The security of the RSA cryptosystem relies on the believed difficulty of factoring large composite integers. About eight sites are attempting to factor RSA-768, a 768-bit challenge number. The best known algorithm is Number Field Sieve, whose current record is 663 bits. Existing software needs upgrades to 64-bit manycore systems. I will describe some proposed algorithmic adjustments as we work to meet this challenge on state-of-the-art hardware.

Preliminary Design of Post-Sieving Processing for RSA-768

Peter L. Montgomery
Microsoft Research
Redmond, WA, USA
Also CWI, Amsterdam

Presented at CADO Integer Factorization Workshop
October, 9, 2008

Factoring and RSA

- RSA cryptosystem chooses two primes p, q, publishing the product $N = pq$.
- Encrypt a message M with $0 \le M < N$ as $(M^e) \bmod N$, typically with $e = 65537$.
- We can recover M easily knowing p and q, but don't know how to get M in polynomial time without this factorization.

RSA-768 Challenge

- A 768-bit composite integer, supposedly with two 384-bit factors.
  - Typifies a public RSA modulus using 768-bit keys.
- Best known algorithm: General Number Field Sieve (GNFS, or simply NFS).
- Present (2008) GNFS record:
  - RSA-200 (200 decimal digits, about 663 bits),
  - Jens Franke et al, May, 2005.
  - http://www.hyperelliptic.org/tanja/SHARCS.

Partial Challenge history

- RSA-100 Apr 1991 MPQS Arjen Lenstra
- RSA-110 Apr 1992 MPQS Lenstra, Mark Manasse
- RSA-120 Jun 1993 MPQS Lenstra et al
- RSA-129 Apr 1994 MPQS Lenstra et al
- RSA-130 Apr 1996 MPQS Lenstra et al
- RSA-140 Feb 1999 GNFS CWI et al (Montgomery)
- RSA-155 Aug 1999 GNFS CWI et al (512 bits)
- RSA-576 Dec 2003 GNFS Jens Franke et al, U. Bonn
- RSA-200 May 2005 GNFS Franke et al, German (663 bits) Federal Office for Information Security
- RSA-768 ???? GNFS

CWI role in RSA-768 project

- Dutch grant for RSA-768, 2008-2012.
- CWI project leader Herman te Riele
  - Centrum voor Wiskunde en Informatica
- Graduate student Andrey Timofeev (Computer Science)
- Arjen Lenstra (Switzerland) and Peter Montgomery (USA) are mentors.
- Much of CWI's NFS implementation is ten years old, back when we did RSA-155.

Number Field Sieve phases — Part I

- Input: A composite positive integer N we want to factor, not a prime power.
- **Polynomial selection** finds distinct polynomials f_1, f_2 with common root m mod N, irreducible over **Z**. Let α_1, α_2 denote complex roots thereof.
  - For RSA-768, degrees are 6 and 1. Neither is monic.
  - RSA-200 used degrees 5 and 1.
- Improving this step made GNFS practical in 1999.

Terminology

- Relation: Integer pair (a, b) with b > 0 and gcd(a, b) = 1.
- Relation is smooth if norm of $a - b\alpha_i$ is smooth in $Q(\alpha_i)/Q$, for both extension fields $Q(\alpha_i)$.
- Ideals in extension $Q(\alpha)$ are (usually) uniquely identified by p and by ratio a/b mod p, where prime p divides norm of $a - b\alpha_i$ in $Q(\alpha)$.
- Singleton: An ideal appearing only once in our data.

Number Field Sieve phases — Part II

- **Sieving** finds smooth **relations** – coprime pairs (a, b) for which both ($a - b\alpha_i$) ideals have smooth norms.
  - RSA-768 sieving started in 2007 and is underway.
- **Filtering** organizes these relations into sets, matching multiple occurrences of a prime ideal, trying to shrink matrix size. Some relations are discarded or replicated.

Number Field Sieve phases — Part III

- **Linear algebra** looks for a subset $\{(a_i, b_i)\}$ of the relations such that both $\prod_i (a_i - b_i \alpha)$ are squares.
  - Prime ideal factorization of product will have only even exponents.
  - Linear algebra problem over GF(2) — need vectors in nullspace of sparse matrix.
  - Ideals for smallest primes (say < 160) can be omitted to reduce density, but we will need extra nullspace vectors to compensate.
    - Norms are “almost” square.
  - Quadratic character tests compensate for powers of units and for omitted ideals.

Number Field Sieve phases — Part IV

- **Square root** takes square roots in $\mathbf{Q}(\alpha_1)$ and $\mathbf{Q}(\alpha_2)$, maps both α_1 and α_2 to m mod N, hopes for nontrivial integer congruence $X^2 \equiv Y^2 \pmod{N}$. Take GCD $(X - Y, N)$.
- If congruence is trivial, or if factorization remains incomplete, repeat this step with different dependency from Part III.

Filter inputs (pruning mode)

- One or more files of (supposedly) smooth relations.
- Duplicate relations allowed.
- Some norm divisors (perhaps primes > 1M) appear alongside (*a*, *b*) on input files. Only ideals for supplied primes will be processed.

Desired filter outputs

- A file (or collection of files) retaining only the useful relations.
  - Remove duplicates (all but one).
  - Recursively remove all relations with a singleton ideal.
  - Optionally, merge when an ideal has frequency 2.
- Saved relation-sets may be output in any order.
- Aim for at most 1% false deletions and 5% false retentions.

Estimated RSA-768 sizes

- Large prime bounds 2^{40} (sieving parameter).
  - $2\pi(2^{40}) \approx$ 82 e9 potential ideals for two polynomials.
- Thorsten Kleinjung estimates 60 billion relations needed from sieving.
  - Fewer than 82e9, since many ideals won't appear.
  - This is 700 times as large as any prior CWI run.
- First filter runs will focus on removing duplicates and singleton ideals, to shrink the data.
  - Do these runs at the site where data is collected.

Huygens

- Supercomputer at SARA, Amsterdam.
  - Several Power6 nodes with 32 core each (2008);
  - A few Power6 nodes with 64 core each (planned).
- 4 gigabytes per core, shared within node.

- Aim to fit on smaller nodes.
  - That is, 32 core, 128 gigabytes.
  - Might also use considerable disk space.
  - Documentation recommends two threads/core.
- Want parallel algorithms.

CWI vs. Huygens

- CWI recently acquired 20+ quadcore x86-64 desktop systems, each with 8 gigabytes.

SARA node 32 core 4 Gbyte/core 128 Gbyte
CWI 4 core 2 Gbyte/core 8 Gbyte

- Budget on CPU usage at SARA, none at CWI.

- Convenient for testing parallel algorithms.

Duplication table (one thread)

- Aim to find repeated (a, b) relations.
- Table has LNG two-byte entries, initially zero.
- LNG = (60 billion)/(thread count) to fill 128-Gbyte node.
- Hash functions $h_1(a, b) \rightarrow [0, \text{LNG}-1]$
- and $h_2(a, b) \rightarrow [1, 65535]$.
- Search (circularly) for $h_2(a, b)$, starting at subscript $h_1(a, b)$. If found, discard latest (a, b). If zero found first, put new entry there.
- Stop inserting when 80% full. Use first 48 billion distinct relations (from all threads).

Duplication pass over relations

- Assume we have hundreds of siever output files.
- Each thread empties its local duplication table.
- Each thread opens its own MYOUT for output.
- Each thread reads relations from some input files:
  - Check for syntax or other errors on relation.
  - If good, forward relation to a slave DSLAVE(*a*, *b*) .
    - Duplicates automatically go to same thread.
  - Meanwhile process data forwarded to us.
    - Check for duplicates. Write non-duplicates to MYOUT.
- End loop.
- CAUTION: Some sievers put *a*, *b*, in decimal, some in hexadecimal. Need consistent hashing.

Start of singleton detection

- Choose HIDEAL function, mapping ideals (*p*, *a*/*b* mod *p*) to 64 bits.
- Set up two (global) frequency tables, each with 240 billion 2-bit entries. Zero first table.
  - Would prefer several local tables.
- Choose $HASH_0$ function mapping HIDEAL values to [0, 240e9 - 1]
  - If 82e9 ideals, then 34% of potential $HASH_0$ values are used.
- Each thread opens a survivors file for output.
- Each thread loops over all relations on its input file:
  - Do syntax checks if not done earlier.
  - Write (line number, $HIDEAL_1$, $HIDEAL_2$, ...) to survivors file.
    - Omit ideals unlikely to be singletons.
  - In first table, accumulate frequency of $HASH_0$ (*hid*) for each *hid* from HIDEAL function, saturating at 3 occurrences.

Later singleton processing

- Start with $j = 0$. Loop until few deletions performed.
- Old frequency table has frequencies of $HASH_j$ (saturated at 3).
- Bump j. Choose $HASH_j$ function on HIDEAL values. Zero new frequency table (other table).
- Each thread reads its survivors file.
  - If any ideals in a relation have frequency = 1 in old (read-only) table, delete relation. Otherwise rewrite in old file and accumulate $HASH_{new\ j}$ (*hid*) frequencies in new (read-write) table.
  - Can mix other strategies, such as deleting a relation having several ideals of frequency only 2.
- At end, use line number information to identify which relations on original input files are retained.
  - Be careful in case an input file has grown since first read.

Combined duplicate/singleton

- Range of DSLAVE extended to include a pass, with pass = 1 two thirds of the time and pass = 2 one third of the time.
- Survivors files have line number, (*a*, *b*) values, and HIDEAL values. Line numbers are in increasing order.
- Each thread reads some siever output files.
  - Where DSLAVE(*a*, *b*) has pass = 1, send DSLAVE(*a*, *b*) thread a pointer to relation. It sets flag telling original thread to retain or discard.
    - 40 billion of 60 billion relations processed now, needing 80 billion bytes in duplication tables (67% full).
  - Where pass = 2, delay duplication check on this (*a*, *b*) until next pass, while frequencies are initialized.
- At end, use line numbers to decide which original data to keep.

- Duplication tables dominate memory on pass = 1. When pass = 2, half of memory has $HASH_0$ frequencies, and half has half-size duplication tables. Thereafter frequencies dominate.

Pruning problems

- Possible problems:
  - Heavy I/O may not parallelize well.
    - Perhaps dedicate some space to I/O buffers.
  - A few free relations are lost (ideal appears to be singleton).
  - Inter-thread communication while updating tables.
    - Who owns what parts of array? When is it read only?
    - Do we send updates to owner of (part of) table, or do them ourself?
  - Need considerable disk space to save filter outputs.
- Subsequent filter runs should stress merges but also delete what we missed (algorithm not in these slides).

Matrix construction — Buildmatrix

- Inputs:
  - Sets of relations organized by filter (merge mode).
- Outputs:
  - Matrix (rows for ideals, columns for sets of relations).
    - Nonzero entries only where an ideal has odd exponent.
    - One file with detailed matrix data, another for summary data such as matrix size and row/column weights.
    - Detail file is organized by columns (i.e., sets of relations).
  - Free relations file
    - Where several ideals have same prime norm.
    - Each free relation becomes a low weight matrix column.
    - Unimportant for RSA-768 — only one prime in 720 qualifies.

Buildmatrix major data structure

- Ideal identification — say R of them.
- Ideals of interest have form $(a - b\alpha, p)$, where p is a prime dividing norm $N(a - b\alpha)$.
  - a/b mod p is a (perhaps projective) polynomial root.
  - Allow 6 bytes for p and 6 bytes for a/b ($p < 2^{48}$).
  - 4 bytes for row number within matrix ($R < 2^{32}$).
  - Identification should include polynomial index.
    - Omit that distinction to reduce table size, possibly skipping an ideal. Can lose only when p divides polynomial resultant.
- Estimated $16R$ bytes for R ideals.
- Estimated R = 250 million for RSA-768

Other big Buildmatrix data

- Column weights — 4 *R* bytes.
- Row weights — 4 *R* bytes.
- Vacancies in ideals table, so it can be a 50% occupied hash table indexed by *p* (separate entries for each *a*/*b*) — 16 *R* bytes.
- Total 40 *R* bytes = 10 gigabytes so far.
- Gaps in memory after realloc and rehash.
  - Early estimates of *R* may be too small.
- Still, far below Huygens node capacity.
  - Might fit on 8 Gbyte CWI desktops.

Buildmatrix sequential algorithm

- Set tables empty.
- Loop over relation-sets from filter.
  - Identify ideals with odd exponents. Insert ideals into tables if new. Adjust net exponent of each ideal within this set.
  - Write indices of odd-exponent ideals for this column to detail file.
  - Adjust summary statistics such as row weights.
- End filter loop.
- Sort ideals table by *p*.
  - Entries for same *p* are already nearby.
- Identify and process free relations.

First parallelism attempt

- When relation-set read from filter output, send set to a slave thread.
  - Individual slaves do separate sets.
- Slave factors ideals and updates tables.
- Slave returns set to master, who writes column to detail file.
  - N.B. Order within detail file must match order of sets received from filter.

Parallelism design flaws

- Unsynchronized table updates, such as:
  - Two slaves updating same row weight.
  - Two slaves inserting entries at same place in hash table (same or different new ideal).
- Many non-local memory references.
  - But good locality while factoring ideal norms.
- What is protocol while enlarging tables?

Parallelism retry

- Master-slave communication blocks
  - Perhaps allocate 5 blocks per slave thread.
  - Each block large enough to hold an output from filter.
  - Slave modifies only these blocks and its own data.
- Sample slave tasks:
  - Convert ASCII decimal data from filter output to binary.
  - Compute and factor norms.
  - Identify all un-omitted (*p*, *a*/*b*) ideals with odd exponent.
    - If ideal is already in global table, supply its location (subscript).
    - Otherwise tell master to insert ideal in table and decide where.
- Slave returns block to master, starts another block.
  - Blocks returned in order received,
- Master finishes block, sends new work to slave.
- Free relations found locally at end (ideals for one *p* are nearby).

Further cautions

- Master must be careful when two slaves request same ideal insertion, to avoid repeat entries.
- Slave may read stale table entry while that entry is being inserted. May need synchronization.
- Master can also be a slave, esp. on 1-core systems.
- When tables enlarged (and data moved), master waits for slaves to return their blocks so it can invalidate old ideal locations and use new locations. Row numbers do not change when ideals move.
- Not all filter outputs take same time to process.
  - Larger relation-sets take longer.
  - Responses from different slaves may arrive out of sequence.
  - On CWI hosts, some cores may be running a screensaver or other desktop application.

Factorrelations

- Supplies primes which divide a norm but were omitted by the siever.
  - Runs after filter, before buildmatrix.
- Omitting divisors reduces file sizes during filtering.
- Optionally, checks that supplied norm divisors are really prime.
- Use algorithms such as trial division and ECM.
  - Small memory.
- Send each relation-set to a slave.
  - Present CWI code uses Pollard Rho.

Square root

- Inputs:
  - Polynomial $f(X)$ irreducible over **Z** with root α.
  - A set $\{(a_i, b_i)\}$ of relations such that

    $P = \prod_i (a_i - b_i \alpha)$ is (almost) a square in **Q**(α).
    - In practice some operands are in a denominator.
  - Integers m, N with $N > 0$ and $f(m) \equiv 0 \pmod N$.
  - Quadratic character tests compensate for powers of units and for omitted ideals during linear algebra.

- Outputs (algorithm invoked twice):
  - Image of sqrt(P) under $\alpha \rightarrow m \pmod N$

Quadratic character phase

- Choose pairs (q_k, r_k) where q_k is prime and $f(r_k) \equiv 0 \pmod{q_k}$ for one f.
- If $P = \prod_{i \in S} (a_i - b_i \alpha)$ is a square, as desired, then $\prod_{i \in S} (a_i - b_i r_k) \bmod q_k$

  is quadratic residue for all k.
- For each "almost" square P (and its S), compute the Jacobi symbols for all k. Do perhaps 100-300 values of k, to account for rows deleted from matrix. Solve small system mod 2.

Faster quadratic characters

- Tiny memory requirements so far.
- Euclidean-like algorithm for Jacobi symbols takes O(log(q_k)).
  - Use smallish primes, say $q_k \approx$ 10000.
  - Use table look-up for (x over q_k)

    when $0 < x < q_k$.
  - Ignore powers of q_k in x when q_k divides x.
  - Error if $x = 0$.
  - 500 arrays each 10000 bits is 1 megabyte.

Parallel quadratic character phase

- Want $\prod_{i \in S} (a_i - b_i r_k) \bmod q_k$ to be squares for all k, many candidate S.
- The (q_k, r_k) pairs are in shared memory.
- As master loops over relation-sets from filter output, it sends each (a_i, b_i) pair to some slave.
- Slave converts a_i and b_i from decimal to binary, computes $(a_i - b_i r_k) \bmod q_k$ for all k, updates local partial products of Jacobi symbols, waits for more from master.
- At end, all slaves combine their results.
- Solve small binary system to get sample S.

Square root sequential — Accumulation phase

- Maintain partial products of
- $P = \prod_i (a_i - b_i \alpha) \bmod f(\alpha)$

- a) Logarithms at complex embeddings;
- b) Modulo some CRT primes exceedng largest prime in relations;
- c) Ideal factorization of principal ideal (P).

Ideal representation

- As in buildmatrix, p and a/b mod p uniquely identify most ideals. 12 bytes for $p < 2^{48}$.
- Exceptional ideals have norm dividing polynomial discriminant and need special treatment even if skipped by buildmatrix.
  - Example: $X^2 + 18$, $p = 3$. Ideals $(\alpha/3 \pm 1, 3)$ are distinct, but $X^2 + 18$ has unique root mod 3.
- Use PARI to distinguish hard cases.
- Each ideal in table needs an exponent (possibly negative).
  - Can limit to ±126 (one byte) if we replicate ideal when exponent is high.

Ideal table size

- Estimated R = 250 million matrix ideals with odd exponent somewhere, when using two polynomials in buildmatrix.
- Square root does polynomials separately.
- Square root code processes all ideals with nonzero (not just odd) exponent.
  - Guess 5(R/2) = 625 million ideals per polynomial.
- At 13 bytes/entry, this is 8 gigabytes.

Parallel square root — Accumulation phase

- Initialize PARI (recent thread-safe version).
- Choose CRT primes (larger than sieving primes).
- Partition ideals table across slaves, using a hash function to decide which slave is responsible for each ideal.
- Set partial products (complex embeddings, principal ideal factorization, mod CRT primes) to 1, on all slaves.
- Loop over relations $\{(a_i, b_i)\}$
  - Send each relation (or a set) to some slave.
  - Slave updates its partial products.
  - Occasionally ship block of ideal exponents to responsible thread, clearing local copy.
    - Each of 64 threads might store up to 1000 ideals for each other thread, This is under a gigabyte.
- At end, merge ideal factorizations and embeddings across all threads.

Sequential square root — Reduction phase

- Input:
  The accumulation data for some principal ideal (Q^2), where Q in $\mathbf{Q}(\alpha)$ is unknown. Also the complex logarithms and CRT embeddings of Q^2.

- Desired output:
  Value of $Q(m)$ mod N (up to sign)

- Algorithm:
  - If logarithms of Q^2 are small and denominator is small then
    - Use CRT to recover Q^2, with small coefficients
    - Take square root in number field
    - Take image modulo N.
  - else
    - Find Q_1 such that Q_1^2 shares many factors with Q^2 .
    - Apply algorithm recursively to Q^2 / Q_1^2
  - end if

Parallel square root — Reduction phase

- Knowing the net exponent of each ideal, partition ideals approximately evenly amongst the slaves.
- Also distribute complex logarithms evenly.
- CRT data need not be moved.

- Each slave applies the sequential algorithm until it cannot improve locally. Everyone's data is multiplied together for the final iterations.

RSA-200 linear algebra

- Block Wiedemann on matrix with 64e6 rows and columns (11e9 nonzero entries).
  - Three months on 80 Opternons for RSA-200.
- Thorston Kleinjung estimates RSA-768 matrix will have R = 250 million rows and columns – four times as many.
  - Comparable density (circa 200 nonzero entries per column).
  - First guess $(250/64)^2$ = 15 times as long.
  - Four years is not acceptable.

Linear algebra tasks

- Runs after buildmatrix, before square root.
- Inputs:
  - Sparse matrix **B** over GF(2), built by buildmatrix.
  - Perhaps 200 nonzero elements per column.
  - Up to1000 more columns than rows.
  - About 250 million rows and columns for RSA-768.
- Outputs:
  - Several (128 or 256) vectors **v** over GF(2) with ***B*****v** = **0**.
  - Nonzero bits v_i identity those i selected in

    $P = \prod_i (a_i - b_i \alpha)$.
- Block Lanczos and Block Wiedemann both worthy of consideration.

Matrix storage

- Almost square, very sparse.
  - About 250 million rows and columns for RSA-768.
  - Perhaps 200 nonzero entries per column.
- Partition matrix into blocks size 65536 x 65556.
  - About 3800 x 3800 blocks (3800 = 250M/65536).
  - 50 billion entries, average 3500 per block.
  - Permute rows and columns, trying to balance block weights.
  - 4 bytes per block entry, to store two 16-bit offsets.
    - Uses about 75% of a 256 Gbyte node.
  - If 250 M estimate is too small, we won't fit.

Applying matrix

- Block Lanczos and Block Wiedemann are iterative. Each needs to apply the matrix to arbitrary binary vectors (actually to 64 or 128 or 256 vectors at a time).
- Block Lanczos lets $\mathbf{A} = \mathbf{B}^T\mathbf{B}$, which is symmetric. Applies both $\mathbf{B}$ and $\mathbf{B}^T$.
- Block Wiedemann appends zero rows to $\mathbf{B}$, getting a square $\mathbf{A}$. Applies only $\mathbf{B}$. Matrix element $a_{i,j}$ is stored on the thread responsible for element i of vectors.

Block Lanczos main processing

- Want solutions to $\mathbf{Av} = \mathbf{0}$, given $\mathbf{A}$.
- $\mathbf{A}$ is symmetric n x n. $\mathbf{v}$ might be n x 128.
- Start with random n x 128 vector $\mathbf{y}$. Compute $\mathbf{Ay}$. Use orthogonality to find an $\mathbf{x}$ with $\mathbf{Ax} = \mathbf{Ay}$.
- $\mathbf{v} = \mathbf{x} - \mathbf{y}$ is in null space of $\mathbf{A} = \mathbf{B}^T\mathbf{B}$. Take linear combinations of columns so it is in null space of $\mathbf{B}$.

Block Lanczos costs

- About *n*/127 iterations.
  - Each iteration applies **A** once.
  - Some inner products each iteration.
  - Also some $\mathbf{v}_1 = \mathbf{v}_1 + \mathbf{v}_2\ \mathbf{c}$ where **c** is 128 x 128.
  - Much communication applying **A**, little elsewhere.
- Five temporary *n* x 128 vectors
  - These use 20 billion bytes (*n* = 250 million).
- Four permutation arrays need 16*n* = 4 billion bytes (accessed only during initialization, checkpoints, and final processing).
- Try to reserve 10% of memory for OS and MPI.

Block Wiedemann

- Try to find minimal polynomial of square matrix **A**.
- Chooses random *n* x 128 vector $\mathbf{v}_0$.
- Repeatedly applies **A** to get $\mathbf{v}_j = \mathbf{A}^j\mathbf{v}_0$ for many *j* .
- Minimal polynomial will have 0 as a root.
- Comparable storage to Block Lanczos.

Fault detection

- Multi-month computations may experience hardware errors.
- Desire to detect errors and restart from earlier state.
- Adi Shamir suggested scheme (2005 presentation to Microsoft Research)).
- Works for first phase of Block Wiedemann but apparently not for Block Lanczos.

Shamir fault detection

- Choose random 1 x n row vector $\mathbf{r}_0$.
- Choose integer k, perhaps 200.
- Denote $\mathbf{r}_j = \mathbf{r}_0 \mathbf{A}^j$ and $\mathbf{w}_j = \mathbf{r}_{j+k}$.
- Compute $\mathbf{w}_0$ on a reliable machine.
- Expect $\mathbf{r}_0 \mathbf{v}_{j+k} = \mathbf{r}_0 \mathbf{A}^{j+k} \mathbf{v}_0 = \mathbf{w}_0 \mathbf{A}^j \mathbf{v}_0 = \mathbf{w}_0 \mathbf{v}_j$.
- At iteration j, sometimes save
  $\mathbf{w}_0 \mathbf{v}_j$ (1 x 128) for comparison with $\mathbf{r}_0 \mathbf{v}_{j+k}$ at iteration $j+k$.

Linear algebra considerations

- Which algorithm is faster? Uses less inter-thread communication?
- Which needs smaller checkpoints?
- Can we adapt Block Lanczos for fault detection? Perhaps test its invariants.
- Can either algorithm be adapted to support discrete logarithms by NFS?