Batch RSA

Amos Fiat*
Department of Computer Science
Tel-Aviv University
Tel-Aviv, Israel

Abstract

Number theoretic cryptographic algorithms are all based upon modular multiplication modulo some composite or prime. Some security parameter n is set (the length of the composite or prime). Cryptographic functions such as digital signature or key exchange require $O(n)$ or $O(\sqrt{n})$ modular multiplications ([DH, RSA, R, E, GMR, FS], etc.).

This paper proposes a variant of the RSA scheme which requires only polylog(n) ($O(\log^2 n)$) modular multiplications per RSA operation. Inherent to the scheme is the idea of batching, *i.e.*, performing several encryption or signature operations simultaneously. In practice, the new variant effectively performs several modular exponentiations at the cost of a single modular exponentiation. This leads to a very fast RSA-like scheme whenever RSA is to be performed at some central site or when pure-RSA encryption (*vs.* hybrid encryption) is to be performed.

An important feature of the new scheme is a practical scheme that isolates the private key from the system, irrespective of the size of the system, the number of sites, or the number of private operations that need be performed.

1 Introduction

Almost all number-theoretic cryptographic schemes in use today involve modular multiplication modulo a composite or prime. Some security parameter n is set, equal to the length of the modulus N. In fact, factoring a composite or solving the discrete log problem can be done in time exponential in $\sqrt{n \log n}$. Thus the *real* security parameter is approximately $\sqrt{n}$. Throughout this paper, N will denote the modulus and $n = \log(N)$.

Irrespective of the scheme, the private operation (decryption, digital signature generation) must depend upon some secret string that is at least as long as the

*Work performed at UC Berkeley and ARL, Israel

G. Brassard (Ed.): Advances in Cryptology - CRYPTO '89, LNCS 435, pp. 175-185, 1990.

security parameter. If the secret information used during the private operation were shorter than the security parameter then the cryptanalyst could guess the secret and break the scheme.

Various schemes for key-exchange, public key cryptosystems, and digital signature have been proposed ([DH], [RSA], [R], [E], [GMR], [FS], ...). In several schemes the secret is used as an exponent during the private operation and therefore the number of modular multiplications required is at least the security parameter. (In fact, many such schemes above require n modular multiplications per private operation, not $\sqrt{n \log n}$). For example, in the RSA scheme it makes no sense to choose a decryption exponent shorter than the security parameter, $(\Omega(\sqrt{n \log n}))$, otherwise guessing this exponent would break the scheme.

The Fiat-Shamir signature scheme [FS] does not use a secret exponent yet it too requires as many multiplications as the security parameter. This is related to the probability that the the prover cheats in the underlying zero-knowledge proof — essentially, every multiplication cuts down the probability of cheating by a factor of two.

In general, it is not true that the public operation (encryption, digital signature verification) requires poly(n) modular multiplications. Various schemes have a fast public operation, *e.g.*, a small encryption exponent for RSA.

The main result in this paper is to circumvent the "lower bound" above, and obtain fast public *and* private operations. We achieve polylog(n) multiplications per private operation, in contradiction to the poly(n) "lower bound". We cannot escape using a long secret string, but the work is averaged over several private operations batched together.

In practice, the problem with performance does not seem to be in a distributed setting but rather with centralized applications. Todays microprocessors can perform hundreds of modular multiplications in a few seconds. Large central mainframes are obviously faster, yet much less cost-effective with respect to processing power.

Many applications require a centralized setting. Several suggested applications of digital signatures are almost irrelevant without a large central clearinghouse, and such a clearinghouse may be required to generate digitally signed receipts in response to transactions. Another typical application is a mainframe that has to decrypt many transactions, (financial data, session initiation key exchange, etc.). The scheme presented here is particularly suitable for such centralized applications.

The underlying idea behind our new scheme is to batch transactions. Rather than perform one full-scale modular exponentiation per digital signature as with RSA, the scheme performs one full-size exponentiation and subsequently generates several independent digital signatures.

Our scheme requires $O(\log^2 n)$ multiplications for a batch size of $n/(\log^2 n)$ messages. We also require up to two modular divisions per signature/decryption — this is a low order term and can be ignored.[1] Clearly one must optimize the batch size

[1] Modular division is equivalent to multiplication for quadratic algorithms ($O(n^2)$ bit operations — *e.g.*, [K] section 4.5.2 problem 35) and equivalent to $O(\log n)$ multiplications asymptotically (*I.e.*, $O(n \log^2 n \log\log n)$ bit operations — [AHU] section 8.10).

for a specific modulus size, and one can obtain *better* results for smaller batches if the modulus is (relatively) small.

Generally, we have a tradeoff between the batch size b and the number of multiplications per signature. Let cn denote the number of modular multiplications required for an n bit exponentiation ($c \approx 1.5$). Given a batch of b messages, $b < n$, we can generate b digital signatures at a cost of $cn/b + O(\log^2 b)$ multiplications per signature (plus two modular divisions). For a fixed batch size k, the work required to generate *all* k signatures is effectively equal to the work required for *one* RSA signature.

Similarly, rather than perform one full-size exponentiation to decrypt an RSA-encrypted block, the new scheme performs one such exponentiation and subsequently decrypts several RSA-encrypted blocks. This is relevant in the context of mainframe decryption (hybrid scheme or pure) and in the context of pure-RSA decryption generally. With respect to a pure RSA encryption scheme, this simply means that the block size is some multiple of the RSA modulus size. We have a tradeoff between block size and time, for larger blocks we spend less time overall.

Another application of the methods presented here is to generate Shamir's cryptographically secure pseudo-random sequence [S] with the same gain in performance. In this context, the block size penalty mentioned above does not occur. It is noteworthy that Shamir himself considered his scheme in [S] impractical due to the great number of multiplications required.

Even if we completely ignore the issue of performance and use full-sized encryption exponents, one important point concerning Batch RSA is that only one root need be extracted, irrespective of the batch size. This is related to the questions posed in [AFK] on computing with and Oracle. Many private operations can be performed by performing *one* private operation. The preliminary work to merge the batch into one problem involves no secret data, neither does the split-up phase after the root extraction. If the private operations involve decryptions then we can ensure security even if the data flow path passes through insecure devices (multiply with a random value whose appropriate root is known). The communications overhead is minimal, one never needs to transmit more than n bits to the next stage (for both merge and split-up phases). Thus, a multi-site multi-mainframe system could store the private key on one weak processor (at a PC on the CEO's desk?), never transmitting more than n bits to and from a site.

2 Background and Central Observation

An RSA digital signature to a message M is simply the e'th root of M modulo N. The public key is the pair (N, e) whereas the private key is the prime factorization of N, e is chosen to be relatively prime to Euler's totient function ϕ of the public key modulus N.

To generate a digital signature on M one first computes $d = e^{-1} \pmod{\phi(N)}$ and then computes

$$M^d \pmod{N} = M^{1/e} \pmod{N}.$$

Thus, every digital signature consists of one full-sized modular exponentiation. ([QC]

suggest the use of the chinese remainder theorem so digital signature generation is slightly faster).

Fundamental to getting polylog(n) multiplications per private operation is the use of (relatively) small encryption exponents for RSA. Using a small encryption exponent means choosing e to be some small constant (say 3), and generating the public key N so that $\phi(N)$ is relatively prime to e. However, choosing a small encryption exponent says nothing about the decryption exponent d. Generally, d will be $\Omega(\phi(N))$. In fact, if d were too small (less than exponential in the security parameter), it would allow the cryptanalyst to attack the scheme. In some sense, we attain the effect of a very small d (length polylog in the security parameter), without compromising security.

Our RSA variant grants some leeway in the value of e. For example, choose two parameters S and R so that S and $R - S$ are small (*e.g.*, $S = n^c$, $R = S + n$.). A public key N is chosen so that $\phi(N)$ is indivisible by all primes in the range $S, \ldots, R$. A valid digital signature is of the form $(s, M^{1/s} \bmod n)$, where s is *any* prime in the range $S, \ldots, R$.

To motivate this variant consider the following example:

Example 2.1:

Given two messages $0 < M_1, M_2 < N$, we wish to compute the two digital signatures $M_1^{1/3} \pmod N$ and $M_2^{1/5} \pmod N$.
Let

$$\begin{aligned} M &= M_1^5 \cdot M_2^3 \pmod N, \\ I &= M^{1/15} \pmod N. \end{aligned}$$

Now, we can solve for $M_1^{1/3}$, $M_2^{1/5}$ as follows:

$$\begin{aligned} \frac{I^6}{M_1^2 \cdot M_2} &= M_2^{1/5} \pmod N; \\ \frac{I}{M_2^{1/5}} &= M_1^{1/3} \pmod N. \end{aligned}$$

Note that we require *one* full-sized exponentiation to compute $I = M^{1/15} \pmod N$ and a constant number of modular multiplications/divisions for preprocessing and to extract the two digital signatures. The rest of this paper is devoted to the generalization of example 2.1.

3 Batch RSA

As above, let N be the RSA modulus, $n = \log_2(N)$, and let b be the batch size.

Let $e_1, e_2, \ldots, e_b$ be b different encryption exponents, relatively prime to $\phi(N)$ and to each other. Choosing encryption exponents polynomial in n implies that their product, $E = \prod_{i=1}^{b} e_i$, is $O(b \log n)$ bits long. Choosing the encryption exponents as the first b odd primes gives us $\log(E) = O(b \log b)$.

Given messages m_1, m_2, ..., m_b, our goal is to generate the b roots (digital signatures/decryptions):

$$m_1^{1/e_1} \pmod N, \quad m_2^{1/e_2} \pmod N, \ldots, \quad m_b^{1/e_b} \pmod N.$$

Let T be a binary tree with leaves labelled $e_1, e_2, \ldots, e_b$. Let d_i denote the depth of the leaf labelled e_i, T should be constructed so that $W = \sum_{i=1}^{b} d_i \log e_i$ is minimized — similar to the Huffman code tree construction. For our main result of $O(\log^2 n)$ multiplications per RSA operation we could simply assume that T is a full binary tree, asymptotically it makes no difference. In practice, there is some advantage in using a tree that minimizes the sum of weight times path length.

Note that $W = O(\log b \log E)$. We will show that the number of multiplications required to compute the b roots above is $O(W + \log N)$.

Our first goal is to generate the product

$$M = m_1^{E/e_1} \cdot m_2^{E/e_2} \cdots m_b^{E/e_b} \pmod N.$$

It is not difficult to see that this requires $O(W)$ multiplications.

Use the binary tree T as a guide, working from the leaves to the root. At every internal node, take the recursive result from the left branch (L), raise it to the power E_R where E_R is the product of the labels associated with leaves on the right branch. Similarly, take the result from the right branch (R) and raise it to the power E_L which is the product of the labels on the left branch. Save the intermediate results L^{E_R} and R^{E_L} (required later). The result associated with this node is $L^{E_R} \cdot R^{E_L}$. The product M is simply the result associated with the root. (See figure 1, the ith leaf is labelled with the ith odd prime).

We now extract the Eth root of the product M:

$$M^{1/E} = m_1^{1/e_1} \cdot m_2^{1/e_2} \cdots m_b^{1/e_b} \pmod N.$$

This involves $O(\log N)$ modular multiplications — equivalent to one RSA decryption.

The factors of $M^{1/E}$ are the roots we require. Our next goal is to break the product $M^{1/E}$ into two subproducts, the breakup is implied by the structure of the binary tree T used to generate the product M. We repeat this recursively to break up the product into its b factors. (See figure 2).

Let $e_1, e_2, \ldots, e_k$ be the labels associated with the left branch of the root of the binary tree T. We define an exponent X by means of the Chinese remainder theorem:

$$\begin{aligned}
X &= 0 \pmod{e_1}, \\
X &= 0 \pmod{e_2}, \\
&\vdots \\
X &= 0 \pmod{e_k}, \\
X &= 1 \pmod{e_{k+1}}, \\
X &= 1 \pmod{e_{k+2}}, \\
&\vdots \\
X &= 1 \pmod{e_b}.
\end{aligned}$$

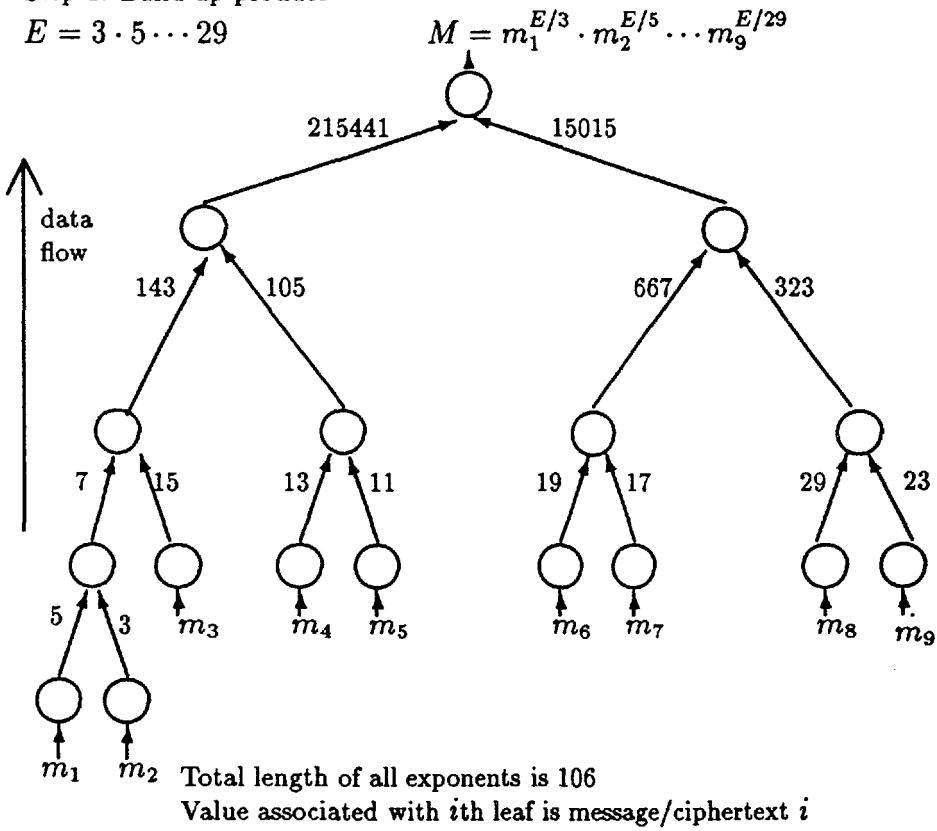

Figure 1: Build up Product and Extract Root

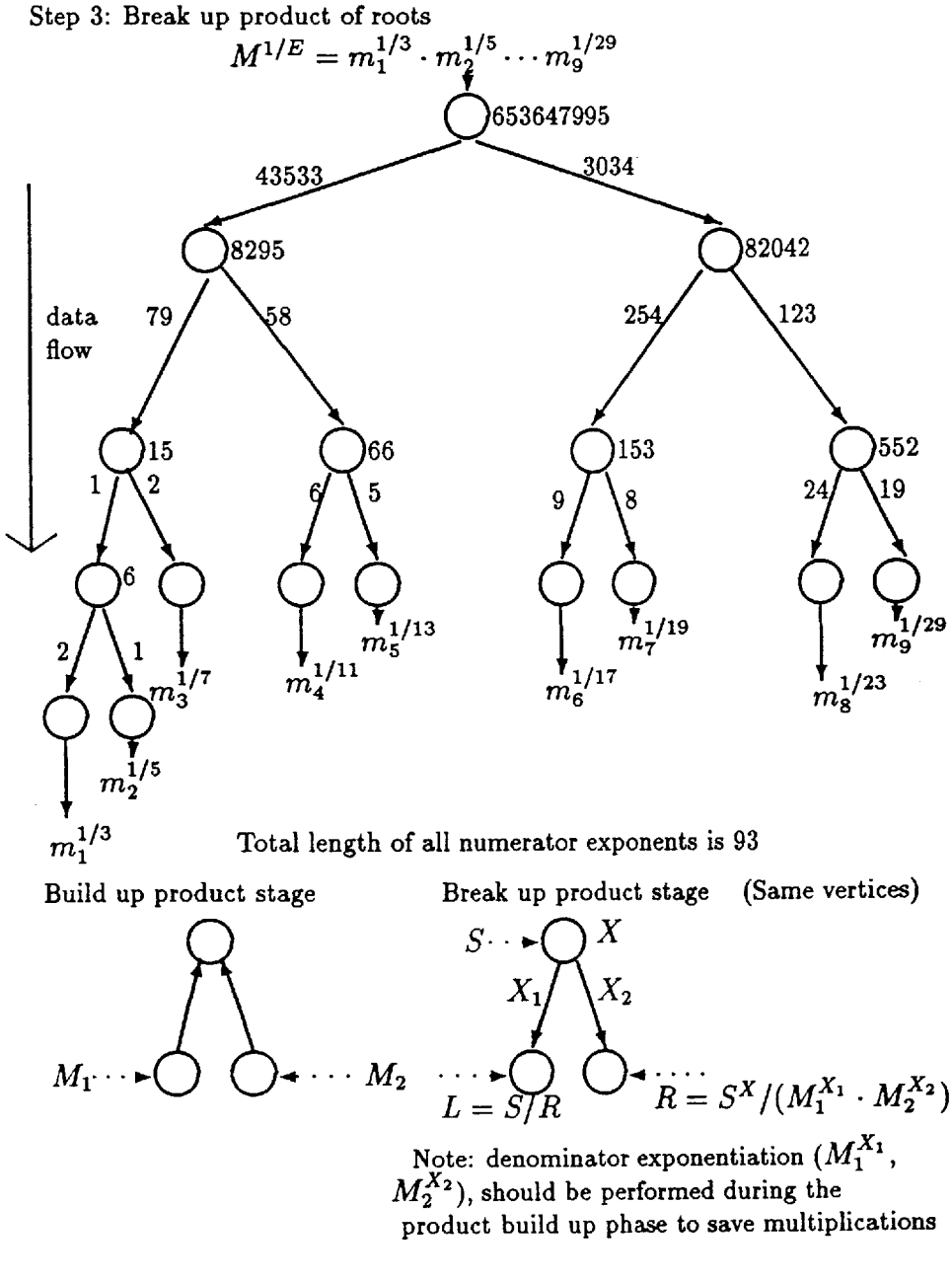

Figure 2: Break up Product of Roots

There is a unique solution for X modulo $\prod_{i=1}^{b} e_i = E$.

By definition

$$X = (\prod_{i=1}^{k} e_i) \cdot X_1,$$
$$X - 1 = (\prod_{i=k+1}^{b} e_i) \cdot X_2.$$

Let $P_1 = \prod_{i=1}^{k} e_i$ and $P_2 = \prod_{i=k+1}^{b} e_i$, then $X = P_1 \cdot X_1$ and $X - 1 = P_2 \cdot X_2$. Note that $\log X < \log E$, $\log X_1 + \log P_1 = \log X$, $\log X_2 + \log P_2 = \log X$, $\log E = \log P_1 + \log P_2$, and thus $\log X_1 + \log X_2 < \log X$.

Denote

$$M_1 = m_1^{P_1/e_1} \cdot m_2^{P_1/e_2} \cdots m_k^{P_1/e_k}, \text{and}$$
$$M_2 = m_{k+1}^{P_2/e_{k+1}} \cdot m_{k+2}^{P_2/e_{k+2}} \cdots m_b^{P_2/e_b}.$$

Note that M_1 and M_2 have already been computed, as the left and right branch values of the root, during the tree based computation of M.

Raise $M^{1/E}$ to the X*th* power modulo N:

$$\begin{aligned}(M^{1/E})^X &= (\prod_{i=1}^{b} m_i^{1/e_i})^X \\ &= (\prod_{i=1}^{k} m_i^{1/e_i})^{P_1 \cdot X_1} \cdot (\prod_{i=k+1}^{b} m_i^{1/e_i})^{P_2 \cdot X_2} \cdot \prod_{i=k+1}^{b} m_i^{1/e_i} \\ &= M_1^{X_1} \cdot M_2^{X_2} \cdot \prod_{i=k+1}^{b} m_i^{1/e_i}.\end{aligned}$$

To solve for $\prod_{i=k+1}^{b} m_i^{1/e_i}$ we raise M_1 to the power X_1, raise M_2 to the power X_2, and divide out. To solve for $\prod_{i=1}^{k} m_i^{1/e_i}$ we divide $M^{1/E}$ by $\prod_{i=k+1}^{b} m_i^{1/e_i}$.

The recursive continuation of this procedure is clear.

Every leaf labelled l contributes $\log l$ bits to the appropriate exponents (X and X_1 or X and X_2) for *every* level between the leaf and the root. Thus, the overall number of multiplications is $O(W)$. The number of modular divisions required is $O(b)$.

To summarize:

Lemma 1 *Let $e_1, e_2, \ldots, e_b$ be b different encryption exponents, relatively prime to $\phi(N)$ and to each other. Given messages m_1, m_2, ..., m_b, we can generate the b roots*

$$m_1^{1/e_1} \pmod N, \quad m_2^{1/e_2} \pmod N, \ldots, \quad m_b^{1/e_b} \pmod N$$

in $O(\log b(\sum_{i=1}^{b} \log e_i) + \log N)$ modular multiplications and $O(b)$ modular divisions.

By choosing the e_i exponents to be polynomial in n and choosing the batch size $b = n/\log^2 n$ we get $O(n)$ multiplications overall and $O(\log^2 n)$ multiplications per root.

Remark: We could choose the encryption exponents to be exponential in polylog(n), for any polylog, and still get polylog(n) multiplications for both encryption and decryption operations.

4 Notes on Security

Use of small encryption exponents for RSA was first suggested by Knuth [K]. A problem arises in the use of small exponents for standard RSA *encryption* if the message is numerically small or messages related via some known polynomials are encrypted to several different recipients. (M. Blum [B] for identical messages, Håstad [H] for fixed polynomials).

We have a more serious problem with encryption in that if the same message is encrypted with different (relatively prime) encryption exponents modulo the *same* modulus then the message can be reconstructed.

In both cases, if RSA is used for key exchange then there is no problem, all the cryptanalyst can learn are random values modulo N. Otherwise, it seems that standard cryptographic practices of randomizing cleartext and appropriate feedback mechanisms effectively overcome these attacks.

One variant of our scheme would be to use different encryption exponents for every encryption or digital signature. *E.g.*, the i*th* prime for the i*th* operation. As long as there are no more than a polynomial number of transactions then both private and public operations would require polylog(n) multiplications.

Shamir has shown that knowing $m^{1/p_1}, m^{1/p_2}, \ldots, m^{1/p_k}$ cannot give us m^{1/p_0} for pairwise relatively prime p_i [S] (as we could extract m^{1/p_0} using this procedure as a black box).

A possible advantage to this one-to-one message-prime relationship is in that it breaks the multiplicative relationship between different RSA blocks. Standard RSA lets you forge any digital signature for messages that are products of previous messages and their inverses.

As the Batch RSA merger and split-up phases involve no secret information, this lets us use Batch RSA to isolate the private key from the system, irrespective of it's size. All private transactions can be reduced to one root extraction that can be solved on a weak and isolated processor. Every link-encryptor, mainframe, etc., requests one root, these roots can be merged again so the entire system is driven by one root extraction. To ensure security in transit, the root requested can be completely random using the standard Zero-Knowledge trick of multiplying the real value with a random value raised to an appropriate power.

5 Constants and Practical Considerations

We assume that modular exponentiation requires $c \cdot k$ modular multiplications for an exponent of length k, $c = 1.5$. This is true for the standard exponentiation algorithm if the exponent is chosen at random. None of our exponents are really random but this is a reasonable upper bound on the work required.

Generating M as described in the proceeding section requires $c \cdot W$ multiplications. Taking the Eth root requires $c \cdot \log(N)$ multiplications, extracting the factors from $M^{1/E}$ requires $2c \cdot W$ multiplications.

In fact, we can do better — extracting the factors of $M^{1/E}$ can be done in $c \cdot W$

multiplications provided that about $W/4$ additional multiplications are done when computing M. Overall, the number of multiplications required to extract the b roots is therefore $2c \cdot W + W/4 + c \cdot \log N$.

Reducing the number of multiplications to extract the factors of $M^{1/E}$ involves a slight digression: Our goal is to compute $y^{Z_1} \pmod N$ and $y^{Z_2} \pmod N$. If Z_1 and Z_2 are random then it seems that this requires $c \cdot (\log Z_1 + \log Z_2)$ modular multiplications. In fact, we can compute $y^{Z_1 \cap Z_2} \pmod N$ where $Z_1 \cap Z_2$ denotes the bitwise and operation between Z_1 and Z_2, compute $y^{Z_1 \cap \overline{Z}_2}$ and $y^{Z_2 \cap \overline{Z}_1}$ and multiply the appropriate results to get y^{Z_1} and y^{Z_2}. It is not hard to see that computing the three intermediate products can be done in $\log N + 3/4 \cdot \log N$ multiplications given that Z_1 and Z_2 are chosen at random in the range $1 \ldots N$. In addition, this can be done without any significant cost in storage other than the area required to hold the three intermediate results.

We can use the trick described in the last paragraph so as to compute the values $M_1^{X_1}$ and $M_2^{X_2}$ as a byproduct of computing M, at a cost of $1/4(\log X_1 + \log X_2)$ multiplications. Recall that the final stage in computing M involves raising M_1 to some exponent R and M_2 to some exponent L. The same holds for all levels of the recursion.

6 Acknowledgements

I am very grateful to David Chaum for the great deal of time he spent introducing Moni Naor and myself to the world of untraceability in Berkeley coffee shops. This work has its origins in Shamir's cryptographically secure pseudo random sequence [S] and in David Chaum's observation that multiples of different relatively-prime roots are problematic in the context of untraceable electronic cash [CFN] as the roots can be split apart.

I with to thank Noga Alon, Miki Ben-Or, Manuel Blum, Gilles Brassard, Benny Chor, Shafi Goldwasser, Dick Karp, Silvio Micali, Moni Naor, Ron Rivest, Claus Schnorr, Adi Shamir, Ron Shamir and Yossi Tulpan for hearing me out on this work.

References

[AFK] Abadi, M., Feigenbaum, J., and Kilian, J., On Hiding Information from an Oracle, Proceedings of the 19th Annual ACM Symposium on Theory of Computing.

[AGCS] Alexi, W., Chor, B., Goldreich, O., and Schnorr, C.P., RSA and Rabin Functions: Certain Parts are as Hard as the Whole, SIAM J. Comput., April, 1988.

[AHU] Aho, A.V., Hopcroft J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[B] Blum, M., Personal communication.

[BM] Blum, M. and Micali, S., How to generate Cryptographically Strong Sequences of Pseudo-Random Bits, SIAM J. Comp., 13, 1984.

[BG] Blum, M. and Goldwasser, S., An Efficient Probabilistic Public Key Encryption Scheme which Hides all Partial Information, Proceedings of Crypto '84.

[CFN] Chaum, D., Fiat, A., and Naor, M., Untraceable Electronic Cash, Proceedings of Crypto '88.

[DH] Diffie, W. and Hellman, M.E., New Directions in Cryptography, IEEE Trans. on Information Theory, Vol IT-22, 1976.

[E] El Gamal, T., A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms, IEEE Transactions on Information Theory, Vol IT-31, 1985.

[FS] Fiat, A., and Shamir, A., How to Prove Yourself: Practical Solutions to Identification and Signature Problems, Proceedings of Crypto '86.

[GMR] Goldwasser, S., S. Micali, and R.L. Rivest, A Secure Digital Signature Scheme, SIAM J. Comput., April, 1988.

[H] Håstad, J., On using RSA with Low Exponent in a Public Key Network, Proceedings of Crypto '85.

[K] Knuth, D., The Art of Computer Programming, vol. 2: Seminumerical Algorithms, 2nd ed., Addison-Wesley, 1981.

[MS] Micali, S., and Schnorr, C.P., Efficient, Perfect Random Number Generators, proceedings of Crypto '88.

[QC] Quisquater, J.–J. and Couvreur, C., Fast decipherment algorithm for RSA public-key cryptosystem, Electronic letters, vol. 18, 1982, pp. 905–907.

[R] Rabin, M.O., Digitalized signatures, in Foundations of Secure Computation, Academic Press, NY, 1978.

[RSA] Rivest, R.L., Shamir, A. and Adleman, L., A Method for Obtaining Digital Signatures and Public Key Cryptosystems, Comm. ACM, Vol. 21, No. 2, 1978.

[S] Shamir, A., On the Generation of Cryptographically Strong Pseudorandom Sequences, ACM Trans. on Computer Systems, Vol. 1, No. 1, 1983.