RSA Public Key Cryptosystem

Zhong Li Ashish Tiwari

November 27, 2000

Abstract

This report describes briefly the current research in both hardware and software for fast evaluation of RSA, which consists of computing powers modulo a large number n. The two aspects namely trying to increase the speed of multiplications and reducing the number of multiplications are considered. We discuss heuristics in literature to reduce the number of multiplications for the computation of a given power and then present a heuristic that most of the times performs better than the best known so far. The second part is a survey on hardware implementations to make multiplications faster.

1 Introduction

The focus of our project has been RSA, one of the most widely used Public Key Encryption methods. An "RSA operation," whether for encrypting or decrypting, signing or verifying, is essentially a modular exponentiation, which can be performed by a series of modular multiplications.

RSA encryption function consists of computing $E = m^e \bmod N$, where $N = pq$ for primes p and q. The decryption step is similar, computing $m = E^d \bmod N$. Here e and N forms the public-key and d forms the private-key. In order to ensure security, each of p, q, N should be *large*, atleast 512-bits long. That forces d to be large. We have a choice for e. Since the keys do not change very frequently, hence we can do some pre-computation to speed-up things!

There are 2 different ways to speed up the calculation of $m^e \bmod N$.

- Reducing the number of multiplications required to compute the modular exponentiation.
- Making the multiplication itself faster.

For RSA, we have e fixed. We use the knowledge of e to pre-determine what will be the shortest sequence of modular multiplications needed to get $m^e \bmod N$.

Section 2 contains definitions and some background material, followed by some related work in section 3. In section 4, we present the algorithm. Results are discussed in section 5. Section 6 contains a survey on hardware implementations of fast modular multiplication algorithms used specifically for RSA. In the appendix, the source code for the program is included.

2 Definition and Notation

An *addition chain* for a given number is a list of numbers having the following properties:

- the first number is 1;
- every number is the sum of two earlier number;
- the given number occurs last in the chain.

We can also define an *addition sequence* for a given set of numbers by changing the last condition to:

- the given numbers occur in the sequence.

We view such a sequence as a series of exponents used to do an exponentiation. The *length* of an addition chain or sequence is the number of elements in the chain, apart from the initial one.

For example, the standard (binary) addition chain [8] for the number 15 has length 6:

$$1, 2, 3, 6, 7, 14, 15$$

There is, however, a chain of length 5 that produces 15:

$$1, 2, 3, 6, 12, 15$$

This means that one can compute x^{15} from x in 5 multiplications.

Naturally we are interested in addition chains with as small a length as possible. Our proposed method produces addition chains with lengths as good as the best known heuristics, and at times it does better.

3 Related Work

Computing the shortest addition chain is an NP-complete problem [9]. [1] was the first attempt to give a heuristic for addition chains. In [1], the author has given a strong heuristic, that still remains the best possible to date, for finding the *smallest addition chain* for a given number n. The heuristic achieves results quite close to the lower bound (Details excluded). The essential idea is to first get the binary representation of n. The binary representation is then split into suitable windows covering all 1's in the binary representation, using some heuristic. Then an addition sequence for the window numbers is constructed.

For example consider $711 = 1011000111_{(2)}$. Suppose we generate the two windows 1011 and 111. The numbers are 11 and 7 respectively. An addition sequence for 7 and 11 will be $1, 2, 3, 4, 7, 11$. Therefore, an addition chain for 711 will be the above mentioned addition sequence followed by, $11, 22, 44, 88, 176, 352, 704, 711$. (Double for each bit position and add the window number for each window). If each window is taken to be of size 1, we get the naive BSM (Binary Square and Multiply) algorithm. Therefore, for 711 we need $5 + 7 = 12$ multiplications.

In 1990, Yacobi [6] used ideas from data-compression to form appropriate windows. In particular he used the Lempel-Ziv algorithm starting from the least significant bit. For the same example as above, he would get 4 windows, which are 1, 011, 11 and 1 (left to right). The addition sequence would be $1, 2, 3$ and the addition chain would that plus 1, 2, 4, 8, 11, 22, 44, 88, 176, 352, 704, 711. This gives 13 multiplications.

Both the above methods give equally good addition chains. It must be remarked here that by choosing special encryption keys (like 3, 17, 65536, or any prime of the form $2^{(}2^n) + 1$, we can perform very efficient encryption because for these numbers we know the optimal solution of the *smallest addition chain problem*, which will be the one BSM give! It is believed that choosing these numbers does not endanger the security of the system. This makes encryption very fast. The decryption key depends on the encryption key and cannot be chosen to be one of these special numbers because otherwise it would be very easy to break the system. So, decryption is much slower and we need to use these heuristics for doing the decryption.

4 Algorithm

The main idea is the same - that of splitting up the binary representation of the number into small windows, followed by finding an addition sequence containing all the window numbers. The algorithm is given here in a very high level language.

Algorithm *Addition-Chain*(N)

1 Initialize $i := n$ where n is the size of N, so that $N[i]$ is the most significant bit of N.

2 Identify the largest string starting from position i which occurs again in N. Make that one window.

3 Skip zeros and let i point to the next 1 in N.

4 Repeat steps 2 and 3 until end of N is reached.

We use *suffix trees* to perform step 2. Suffix tree is a standard data structure used in many string algorithms, especially those for substring search. See [**?**] and *string* for details. In our case, the suffix tree is a binary tree, the right edge being a one and the left edge being a zero. Each suffix of the given number (n of them for a n-bit number) that starts with a 1 is stored as a leaf in the tree such that the path from the leaf to the root represents that suffix.

Using the same example, if $N = 711 = 1011000111_{(2)}$, then the windows we get are, 10, 11, 11 and 1. The addition sequence would be (1, 2, 3). The addition chain for 711 is now easily obtained to be 2, 4, 8, 11, 22, 44, 88, 176, 352, 355, 710, 711.

The above algorithm was further optimized to the following,

1 Same as above.

2 Same as above.

2.1 If the size of the detected window is larger than some constant, mark all occurrences of that window substring in the given number, i.e., we mark more than one windows in one step (but all of the marked windows have the same number in them).

3 Skip zeros and all marked windows and let i point to the next free 1.

4 Repeat steps 2, 2.1 and 3 until the end of N is reached.

For a given size of N, we can find a good choice of the constant used in step 2.1 above. In the following results, the column marked "SUF" contains the results for the first algorithm and the one marked "OSUF" has results for the second optimized version of the first algorithm.

5 Results

The algorithm in [6] was implemented. Along with it we also implemented the naive BSM method and two very simple heuristics. All of these algorithms were run on 10 random 512-bit numbers. The results are shown below.

BSM	HEU1	HEU2	ZIP	SUF	OSUF
772	652	648	634	634	624
766	636	628	617	621	615
765	648	642	632	629	622
773	653	645	634	631	627
782	642	640	635	634	631
768	646	637	632	644	625
769	642	638	624	634	629
774	643	639	634	639	618
753	658	648	635	645	633
793	643	635	634	636	625

The results clearly indicate that even very simple heuristics can reduce the number of multiplications required drastically over the naive way of doing it. The proposed algorithm gives better results most of the times. Since these algorithms need to be run just once, a good idea would be to run all the known heuristics and choose the best addition chain. Also, each of this algorithm runs in polynomial time, they are all very fast. This way we will save the most.

6 Making Multiplication Faster

The way modular exponentiation is done is as follows,

- Exponentiation is performed as a series of modular multiplications.
- Modular multiplications can be done in either of 2 ways -
  - Standard multiplication followed by modular reduction.
  - Overlapping the multiplication and the modular reduction operation.
- Multiplication is implemented as a series of addition.

In the recent years the focus has been on making modular multiplication faster rather than reducing the number of multiplications itself [4]. There is a lot of work going on in this area. For a survey, see [7]. The essential idea to note is that the operations in the RSA encryption and decryption algorithm can be very deeply pipelined. Different papers try to achieve the same goal of building a good pipelined architecture in different ways.

The specific question is how to design specialized hardware that would find an X such that $X = AB(\mod N)$ where A, B and N are large numbers (say 512 bit long each). Now, the special property that is exploited is that in RSA, the N does *not* change.

In [5], the authors describe an approach to do modular multiplication in which the multiplication and reduction are separate tasks, the output of first feeding into the input of second. Note that,

$$P \bmod N = \sum_{i=1}^{i=2n} p_i(2^{i-1} \bmod N)) \bmod N$$

So that if $P = [p_1, p_2, \ldots, p_{2n}]$, $p_i \in 0, 1$ is the product of two numbers in binary, then reduction is simply a conditional sum of powers of 2 reduced modulo N. This is referred to as sums-of-residues (SOR) reduction. If we do not perform the outermost modular operation, we may get overflowing in the next exponentiation step. So the authors first find the upper bound for the overflow that can can occur. The results show that for a typical input bit-length of 256 bits, the actual hardware is required to handle 266 bit numbers.

One simple idea that comes to mind is that of storing the moduli of all powers of 2 (modulo N). But the data paths for the lookup table would be very wide. So, in [5] the authors give a way to *calculate* these values while doing the other addition operations, using the following formula,

$$r_i = 2r_{i-1} \text{ iff } (2r_{i-1} - N) < 0$$

$$r_i = 2r_{i-1} - N \text{ iff } (2r_{i-1} - N) \geq 0$$

By storing the result of the comparison, the r_i can be calculated while performing other operations. Based on this idea the authors give a highly pipelined architecture that receives bits from the multiplication unit , lsb first, and calculates the desired sum of residues by calculating the different bits of the residues in different cycle and adding them appropriately. The different pipelined units work on different bits in any one clock cycle. For details, see [5]. Since we can get rid of all bit testing and conditional branches, the algorithm runs extremely efficiently on highly pipelined processors, or RISC machines, since no instruction queue flushing would be required. The authors expect an improvement of approximately 50%. The work in [5] is the subject of a patent application.

The other prevalent approach to perform fast modular multiplication is based on combining multiplication and residue calculation. Instead of working with radix 2, the trend has been to present algorithms based on higher radix $r \geq 4$ that would multiply modulo N in $\log(N)/\log(r)$ clock pulses. The essential step there is,

$$R^{(k-1)} = rR^{(k)} + b(k)A - c(k)N$$

where $R^{(k)}$ holds the partial results of residue calculations ($R^{(0)}$ would be the answer), A, B are the numbers being multiplied, $b(k)$ represents $\log(r)$ bits of B at the proper position and $c(k)$ is usually estimated so that the resulting $R^{(k-1)}$ is bounded above. For more details, see [4].

7 Summary

We have presented an heuristic algorithm for solving the addition chain problem that does as well as the best ones available in the literature. In the second part, there is a survey on developing specialized hardware for making modular multiplication faster. Due to the patent issues, most papers we found in the literature did not give full details of the algorithms being used. Only brief outlines were provided.

References

[1] J. Bos and M.Coster, *Addition Chain Heuristics*, Advances in Cryptology - Proceedings of Crypto '89 (G. Brassard. ed.), LNCS, vol 435, pp 400-407.

[2] Peter de Rooij, *Efficient Exponentiation using Precomputation and Vector Chain Additions*, Advances in Cryptology - Eurocrypt '94, LNCS.

[3] E.F. Brickell, D.M. Gordon, K.S. McCurley, D.B. Wilson, *Fast Exponentiation with Precomputation*, Advances in Cryptology - Proceedings of Eurocrypt '92, LNCS, vol 658, 1993, pp 200-207.

[4] Hikaru Morita, *A Fast Modular-multiplication Algorithm based on a Higher Radix*, Advances in Cryptology - Proceedings of Crypto '89 (G. Brassard. ed.), LNCS, vol 435, pp 387-399.

[5] P.A.Findlay and B.A.Johnson, *Modular Exponentiation Using Recursive Sums of Residues*, Advances in Cryptology - Proceedings of Crypto '89 (G. Brassard. ed.), LNCS, vol 435, pp 371-386.

[6] Y. Yacobi, *Exponentiating Faster With Addition Chains*, Advances in Cryptology - Proceedings of Eurocrypt '90 (I.B. Damgard ed.), LNCS, vol 473, pp 222-229.

[7] Ernest F. Brickell, *A Survey of Hardware Implemetations of RSA*, Advances in Cryptology - Proceedings of Crypto '89 (G. Brassard. ed.), LNCS, vol 435, pp 368-370.

[8] Knuth, D.E., *The art of computer programming*, Vol. 2, Seminumerical algorithms, Addison-Wesley, Reading, Mass., (1969), [[. 398-419.

[9] Downey P., Leony B. and Sethi R, *Computing sequences with addition chains*, Siam Journal of Computing 3 (1981), 638-696.

A Source Code in C

```
# include <stdio.h>

# define SAMPLE 10  /* No of inputs on which program is run */
# define SIZE 512  /* No of bits in the number */
# define SEED 132  /* Random number seed */
# define LIMIT 12  /* The constant parameter in step 2.1 */

typedef struct stree{ /* The node of the suffix tree */
int from;
int to;
int lfrom, lto; /* Left tree string */
int rfrom, rto; /* Right tree string */
short mark;
short lorr;
struct stree* parent;
struct stree* left;
struct stree* right;
} stree;

typedef struct LINKED_LIST {
long num;
struct LINKED_LIST *next;
} llist;

int number[SIZE]; /* The number N */
int multiplications = 0;
stree* tree; /* Suffix tree root */
stree* where[SIZE];
stree* mark[SIZE];
```

```
/* Create and initialize a node of the suffix tree */
stree* init()
{
 stree* temp;

 temp = (stree*)malloc(sizeof(stree));
 temp->from = (temp->to = 0);
 temp->lfrom = (temp->lto = 0);
 temp->rfrom = (temp->rto = 0);
 temp->mark = 0;
 temp->parent = NULL;
 temp->lorr = -1;
 temp->left = (temp->right = NULL);
 return(temp);
}

/* Insert a number inb the suffix tree and return the new tree */
/* The inserted number starts from position f and goes till t in number */
stree* build_suffix_tree(stree* tree, int* number, int f, int t)
/* Put string from i to j in number in the stree */
{
 stree* temp, *tmp;
 int i, j;
 int st = f;
 int lfrom, lto;

 if (t < f) return(tree);
 temp = tree;

 if ((!number[f]) && (temp->left)) {
lfrom = temp->lfrom; lto = temp->lto;
for (i = lfrom, st = f; (i <= lto)&& (st <= t); i++, st++)
if (number[st] != number[i]) break;
if ((i > lto)&&(st<=t)) {
tmp = build_suffix_tree(temp->left,number,st,t);
tmp->from = f;
tmp->to = t;
return(tmp);
}
else if ((i <= lto)&&(st > t)) {
temp->lto = (i-1);
tmp = init();
```

```
if (!number[i]) {
tmp->lfrom = i; tmp->lto = lto;
tmp->left = temp->left;
tmp->left->parent = tmp;
tmp->left->lorr = 0;
tmp->lorr = 0;
}
else {
tmp->rfrom = i; tmp->rto = lto;
tmp->right = temp->left;
tmp->right->parent = tmp;
tmp->right->lorr = 1;
tmp->lorr = 1;
}
temp->left = tmp;
tmp->parent = temp;
tmp->from = f; tmp->to = t;
return(tmp);
}
else if ((i <= lto)&&(st <= t)) {
temp->lto = (i-1);
tmp = init();
if (!number[i]) {
tmp->from = temp->left->from;
tmp->to = temp->left->to - (lto - i + 1);
tmp->lfrom = i; tmp->lto = lto;
tmp->rfrom = st; tmp->rto = t;
tmp->right = init();
tmp->right->parent = tmp;
tmp->right->lorr = 1;
tmp->left = temp->left;
tmp->left->parent = tmp;
tmp->left->lorr = 0;
temp->left = tmp;
tmp->parent = temp;
tmp->lorr = 0;
tmp->right->from = f; tmp->right->to = t;
return(tmp->right);
}
else {
tmp->from = temp->left->from;
tmp->to = temp->left->to - (lto - i + 1);
```

```
tmp->rfrom = i; tmp->rto = lto;
tmp->lfrom = st; tmp->lto = t;
tmp->right = temp->left;
tmp->left = init();
tmp->left->parent = tmp;
tmp->left->lorr = 0;
tmp->right->parent = tmp;
tmp->right->lorr = 1;
temp->left = tmp;
tmp->parent = temp;
tmp->lorr = 0;
tmp->left->from = f; tmp->left->to = t;
return(tmp->left);
}
}
else { /* i > lto && st > t */
return(temp);
}
 }
 else if (number[f]&&(temp->right)) {
lfrom = temp->rfrom; lto = temp->rto;
for (i = lfrom, st = f; (i <= lto)&&(st <= t); i++, st++)
if (number[st] != number[i]) break;
if ((i > lto)&&(st<=t)) {
tmp = build_suffix_tree(temp->right, number,st,t);
tmp->from = f; tmp->to = t;
return(tmp);
}
else if ((i <= lto)&&(st > t)) {
temp->rto = (i-1);
tmp = init();
if (!number[i]) {
tmp->lfrom = i; tmp->lto = lto;
tmp->left = temp->right;
tmp->left->parent = tmp;
tmp->left->lorr = 0;
}
else {
tmp->rfrom = i; tmp->rto = lto;
tmp->right = temp->right;
tmp->right->parent = tmp;
tmp->right->lorr = 1;
```

```
}
temp->right = tmp;
tmp->parent = temp;
tmp->lorr = 1;
tmp->from = f; tmp->to = t;
return(tmp);
}
else if ((i <= lto)&&(st <= t)) {
fflush(stdout);
temp->rto = (i-1);
tmp = init();
if (!number[i]) {
tmp->from = temp->right->from;
tmp->to = temp->right->to - (lto - i + 1);
tmp->lfrom = i; tmp->lto = lto;
tmp->rfrom = st; tmp->rto = t;
tmp->right = init();
tmp->left = temp->right;
tmp->left->parent = tmp;
tmp->left->lorr = 0;
tmp->right->parent = tmp;
tmp->right->lorr = 1;
temp->right = tmp;
tmp->parent = temp;
tmp->lorr = 1;
tmp->right->from = f; tmp->right->to = t;
return(tmp->right);
}
else {
tmp->from = temp->right->from;
tmp->to = temp->right->to - (lto - i + 1);
tmp->rfrom = i; tmp->rto = lto;
tmp->lfrom = st; tmp->lto = t;
tmp->right = temp->right;
tmp->left = init();
tmp->right->parent = tmp;
tmp->right->lorr = 1;
tmp->left->parent = tmp;
tmp->left->lorr = 0;
temp->right = tmp;
tmp->parent = temp;
tmp->lorr = 1;
```

```
tmp->left->from = f; tmp->left->to = t;
return(tmp->left);
}
}
else { /* i > lto && st > t */
return(temp);
}
 }
 else if (number[f]&&!(temp->right)) {
temp->right = init();
temp->right->parent = temp;
temp->right->lorr = 1;
temp->rfrom = f;
temp->rto = t;
temp->right->from = f; temp->right->to = t;
return(temp->right);
 }
 else {
temp->left = init();
temp->left->parent = temp;
temp->left->lorr = 0;
temp->lfrom = f;
temp->lto = t;
temp->left->from = f; temp->left->to = t;
return(temp->left);
 }
}

/* Generate a random number of size SIZE */
void get_number()
{
 int i;

 for (i = 0; i < SIZE; i++)
number[i] = (lrand48()%2);
}

/* Convert a string of 0-1 to a decimal number */
int gnumber(stree* tmp, int* number)
{
 long t = 0;
 int k;
```

```
 for (k = tmp->from; k <= tmp->to; k++)
t = 2*t + number[k];
 return(t);
}

/* Insert a node in a linked list sorted by decreasing order */
llist* insert(llist* chain, long t)
{
 llist* tmp, *back, *new;

 if (t == 1) return(chain);
 back = NULL;
 tmp = chain;
 while (tmp) {
if (t == tmp->num) return(chain);
if (t > tmp->num) break;
back = tmp;
tmp = tmp->next;
 }
 new = (llist*) malloc(sizeof(llist));
 new->num = t;
 new->next = tmp;
 if (!back) return(new);
 back->next = new;
 return(chain);
}

/* Generate a addition chain containing all items in the list */
/* Return the length of the addition chain */
int addition_chain(llist* chain)
{
 llist* tmp;
 int ans = 0, temp, temp1, temp2, temp3;

 tmp = chain;
 while (tmp && tmp->next) {
temp = tmp->num % tmp->next->num;
temp1 = tmp->num / tmp->next->num;
temp2 = temp1 * tmp->next->num;
temp3 = 2 * tmp->next->num;
if (temp > 1) {
```

```
  ans++;
tmp = insert(tmp->next, temp);
}
else if (temp == 1) {
tmp = tmp->next;
ans++;
}
else if (temp == 0) {
tmp = tmp->next;
}
else {
tmp = tmp->next;
printf("ERROR\n");
}
switch (temp1) {
case 0 : printf("Error\n");
 break;
case 2 : ans++;
 break;
case 3: case 4 : ans += 2;
 break;
case 1 : break;
default : if (temp) tmp = insert(tmp, temp2);
 else {
tmp = insert(tmp, temp3);
 }
 break;
}
 }
 if (!tmp) return(ans);
 if (tmp->num == 1) return(ans);
 switch(tmp->num) {
case 0:case 1 :
return(ans);
break;
case 2: return(ans+1);
case 3: case 4 : return(ans+2);
case 5: case 6 : case 8: return(ans+3);
case 7: case 9: case 10: case 12: return(ans + 4);
case 11: case 13: case 14: case 15 : return(ans+5);
default : printf("ERROR\n");
 }
```

```
 return(0);
}

/* Mark all the windows in the number */
int updatenextm(stree* node, int* nextm, int span)
{
 int i, ans = 0;
 if (!node) return(0);
 else {
for (i = node->from; i <= node->from+span; i++) if (nextm[i] <= node->from+span) brea
if (i > node->from+span) {
ans++;
for (i = node->from; i >= 0; i--) if (nextm[i] <= i) break;
for (i = i + 1; i <= node->from+span; i++)
nextm[i] = node->from;
}
ans += updatenextm(node->left, nextm, span);
ans += updatenextm(node->right, nextm, span);
 }
 return(ans);
}

/* Main procedure */
main()
{
 int i, j, k;
 int l, one, inc;
 int howmany = 0;
 int nextm[SIZE];
 int param;
 int value = 5*SIZE;
 stree* temp;
 llist* chain = NULL;

 srand48(SEED);
 for (l = 0; l < SAMPLE; l++) {
 get_number();
 value = 5*SIZE;
 for (param = 0; param < LIMIT; param++) {
 tree = NULL;
 chain = NULL;
 tree = init();
```

```
 for (i = 0; i < SIZE; i++) nextm[i] = 5*SIZE;
 for (i = 0; i < SIZE; i++) if (number[i]) break;
 multiplications = 0;
 one = i;

 for (i = SIZE - 1; i >= one; i--) if (number[i]) {
temp = build_suffix_tree(tree, number, i, SIZE-1);
where[i] = temp;
 }

 i = one; j = 0; inc = 0;
 while (i < SIZE-1) {
temp = where[i];
if (!temp->mark) temp = where[i]->parent; /* Bus error core dump? */
while(temp) {
if (temp->mark) break;
if ((temp->to - temp->from) < (nextm[i] - i)) {
mark[j] = temp;
break;
}
temp = temp->parent;
}
if (i != one) multiplications += (temp->to - temp->from + 2);
if (!temp->mark) {
mark[j]->mark = 1;
j++;
if (temp->to - temp->from < param) {
howmany = 1;
for (inc = temp->from; inc <= temp->to; inc++) nextm[i] = i++;
}
else {
howmany = updatenextm(temp, nextm, temp->to - temp->from);
}
}
else { /* Do nothing ? */
for (inc = temp->from; inc <= temp->to; inc++) {
nextm[i] = i++;
}
howmany = 1;
}
multiplications += (howmany - 1) * (temp->to - temp->from + 2);
```

```
inc = 1;
while (inc) {
inc = 0;
while ((i < SIZE) && (nextm[i] <= i)) i++;
while ((i < SIZE) && !number[i]) {
i++;
multiplications++;
inc = 1;
}
}
 }
 if (i == SIZE-1) multiplications += 2;
 else if (i < SIZE - 1) multiplications += (SIZE - i + 1);

 for (i = 0; i < j; i++) {
chain = insert(chain, gnumber(mark[i], number));
 }
 if (chain) chain = insert(chain, 2);
 multiplications += addition_chain(chain);

 free(tree);
 free(chain);
 if (multiplications < value) value = multiplications;
 }
 printf("No of multiplications required = %d\n", value);
 }
}
```