Defeating modexp side-channel attacks with data-independent execution traces

Torbjörn Granlund*

Abstract

We present an efficient algorithm for computing $m^d \bmod N$, which is resilient to common side-channel attacks. For any two sets of n-bit operands, the algorithm performs the same sequence of operations and yields the exact same memory access traces. It is side-channel silent under reasonable assumptions of the underlying hardware's side-channel silence for basic word operations. We have implemented the algorithm as part of the GMP library, and show that it is almost as efficient as corresponding performance-optimised modular exponentiation code.

1 Introduction

The modular exponentiation operation $m^d \bmod N$, also known as *modexp*, is critical in several cryptographic systems, such as RSA, DSA, and Diffie-Hellman. It is critical in the sense that the cryptosystems' performance depends on the performance of modexp, but also critical in the sense that the security depends on secure implementation of modexp.

In the various cryptographic uses of modexp, some subset of m, d, and N are secret.

In RSA encryption, only m—used for the message—is secret, while the encryption exponent and N are part of the public key. If one uses a modexp implementation that leaks m, then a single secret message is lost.

In RSA decryption, the encrypted message is the base m, the decryption key is the exponent d, while N is the product of two secret primes p and q. One typically computes two separate modexps, one $\bmod p$ using the exponent $d_p = d \bmod (p-1)$, and analogously for q. If one uses a modexp implementation that leaks d then an adversary will be able to decrypt any future message; and if p or q are leaked, then d can be computed from the public key, again allowing the adversary to decrypt any message.

The conclusion is that neither m, d, or N should be leaked by a good modexp implementation.

Unfortunately, many modexp implementations do leak *side-channel* information; if the plain binary *square-and-multiply* algorithm is used, the execution pattern will depend on the exponent d; each 1-bit will cause a square-multiply-square sequence to be performed, while two consecutive squaring operations indicate a 0 bit. Even less naive algorithms, such as the k-ary modexp, tend to leak side-channel information. Practical modexp attacks have been demonstrated by Kocher [6], and Brumley-Boneh [1], as well as other researchers.

In this article, we present an algorithm and implementation for computing $m^d \bmod N$, where m and N are n-bit numbers. We show that the algorithm has certain properties that avoid leakage of side-channel information, such as data-independent execution timing or data-independent cache pattern behaviour. The methods we use for silencing side-channel leakage have some execution time cost, but on our test platforms and using GMP [3] it is only about 20%.

1.1 Side-channel attacks

There are many different types of side-channel attacks. We discuss the following 5 types in this paper:

*T. Granlund is with the Theoretical Computer Science group at the Royal Institute of Technology (KTH), Stockholm, Sweden. Granlund's work was sponsored in part by the Swedish Foundation for Strategic Research. Email: tege@gmplib.org

In a *timing attack* one watches the time of an entire cryptographic operation, or part thereof, in order to statistically determine the cryptographic key.

A *power monitoring attack* watches the consumption of electrical power during a cryptographic operation. This can be particularly effective against RSA, which compute $m^d \bmod N$, and where d is the cryptographic key. In the basic square-and-multiply algorithm, consecutive square operations mean that the key has a 0-bit, while a square-multiply sequence means the key has a 1-bit. It might be the case that multiplication and squaring take slightly different amount of power.

In an *electromagnetic attack* one watches—perhaps at some distance—the electromagnetic radiation or magnetic field from the computer performing the cryptographic operation. This attack can be seen as a refinement of the power monitoring attack, and is probably more realistic. The square-and-multiply algorithm is particularly susceptible to electromagnetic attacks, as multiplication and squaring typically emits slightly different fields, and furthermore will the time required for the two operation usually be different. Practical attacks using an antenna have been demonstrated, where the key can be extracted from a single operation.

In a *cache attack* one watches the effects of CPU cache hit patterns. Any algorithm that uses tables, and where table entries are chosen using sensitive data (key and/or secret message) is susceptible to this attack. The amount of leaked information depends on where the attacker is located; on the same time-sharing system an attacker might set the cache in partially well-defined states, and cause a time-shared cryptographic computation on the same system to behave enough differently to leak useful information. The main leakage channel will be timing variations, but power fluctuations, or electromagnetic radiation variations may also occur.

In a *branch prediction attack* one watches the effects of a CPU's branch prediction hardware. This is similar to a cache attack, and typically also works best in a time-shared system setting.

2 Algorithms for modexp

Most algorithms for modexp rely on decomposing the exponent into a base-2^k number. The simplest form uses base 2, and operates on the exponent bits from the most significant end ("left-to-right" square-and-multiply exponentiation).

```
MODEXP(m, d, N)
    INPUT: m, d = (d_{ℓ-1} d_{ℓ-2} ··· d_1 d_0), N
    OUTPUT: m^d mod N
1   r ← 1
2   for j ← ℓ, ..., 0
3       r ← r^2 mod N
4       if d_j ≠ 0                          // exponent bit set?
5           r ← r · m mod N
6   return r
```

Algorithm 1: Simple modexp algorithm, leaking information as a result of the conditional execution.

The critical operations in modexp are n-bit multiplication and reductions mod N. Many of the multiplications are actually squarings, which can either be performed as plain multiplications, or using special, faster squaring code.

In Algorithm 1 we need $\lfloor \log_2 d \rfloor$ squaring operations, or roughly one per bit in the exponent, and we need one multiply operation for each 1-bit in the exponent, or about $(\log_2 d)/2$ for a random exponent. For each squaring or multiply we reduce the result to a (principal) residue $\bmod N$.

We can save most of the multiplications by decomposing the exponent into base 2^k for a slightly larger k, pre-computing a table of m^ℓ for $0 \le \ell < 2^k$. The powering loop then takes k bits at a time from the exponent, and uses these bits as an index in the pre-computed table. The number of multiplications becomes $\lceil n/k \rceil$ for the actual powering plus $2^k - 1$ for pre-computing the table; the number of squarings decreases from $n - 1$ to $n - k$, a very tiny improvement.

MODEXP(m, d, N)

```
    INPUT: m, d = (d_{ℓ-1} d_{ℓ-2} ··· d_1 d_0), N, k
    OUTPUT: m^d mod N
 1  tab[0] ← 1
 2  for j ← 1, ..., 2^k − 1
 3      tab[j] ← tab[j − 1] · m mod N
 4  r ← 1
 5  j ← ℓ − 1                          // index to most significant exponent bit
 6  while j ≥ 0
 7      b ← (d_j ··· d_i) where i = max(j − k + 1, 0)
 8      for s ← 1, ..., j − i + 1
 9          r ← r^2 mod N
10      r ← r · tab[b] mod N
11      j ← i − 1
12  return r
```

Algorithm 2: Basic k-ary modexp. At the cost of computing and tabling 2^k powers of m, we save most multiplications in the powering loop. There is no conditional execution, resulting in less side-channel leakage.

In the *k-ary sliding-window* algorithm, one pre-computes only odd powers of m, and extracts bit blocks from the exponent where the least significant bit is 1, and furthermore handles exponent bit sequences with only zeroes specially, with just squarings (hence the term *sliding*).

MODEXP(m, d, N)

```
    INPUT: m, d = (d_{ℓ-1} d_{ℓ-2} ··· d_1 d_0), N, k
    OUTPUT: m^d mod N
 1  tab[0] ← m
 2  for j ← 1, ..., 2^{k-1} − 1
 3      tab[j] ← tab[j − 1] · m^2 mod N
 4  r ← 1
 5  j ← ℓ − 1                          // index to most significant exponent bit
 6  while j ≥ 0
 7      b ← (d_j ··· d_i) where i is minimal, j − i + 1 ≤ k and d_i = 1
 8      for s ← 1, ..., j − i + 1
 9          r ← r^2 mod N
10      r ← r · tab[(b − 1)/2] mod N
11      j ← i − 1
12      while j ≥ 0 ∧ d_j = 0
13          r ← r^2 mod N
14          j ← j − 1
15  return r
```

Algorithm 3: Sliding k-ary modexp. The algorithm uses a table of just 2^{k-1} odd powers of m, half compared to plain k-ary modexp. The sliding operation (lines 13–15) further reduces the average number of multiplications.

It is also possible to make a more advanced analysis of the exponent, for further small improvements. For details, refer to [7]. If $N = pq$ for prime p, q and p, q are available, one should usually compute $m^{d \bmod (p-1)} \bmod p$ and $m^{d \bmod (q-1)} \bmod q$ and obtain $m^d \bmod N$ using the Chinese remainder theorem. Our algorithm is applicable for these two modexp operations. In this article, we consider the case where N is prime or is composite but its factors are not available.

3 Side-channel leakage of the algorithms

There are several ways in which the presented modexp algorithms leak into side-channels. We divide our exposition into leakage in the high-level modexp algorithms, here called macro leakage, and leakage arising from the underlying arithmetic operations.

3.1 Macro leakage

In Algorithm 1 there is a conditional execution of the statement $r \leftarrow r \cdot m \bmod N$. This leaks branch prediction information, and instruction cache information. Worse, when using special squaring code for the operation on line 3 (as one should for performance!) there will be an easily measurable pattern of squarings and multiplies; an electromagnetic attack or a power attack could observe the exponent from the pattern of multiplications and a squarings; consecutive squarings implies an exponent 0-bit and a multiply-squaring sequence implies an exponent 1-bit.

From a side-channel silence perspective, Algorithm 1 is really poor.

Algorithm 2 has no conditional execution; for a given n the same execution paths are followed. However, table entries are chosen from exponent bit blocks, leading to an exponent-induced data cache state. An attacker with access to the system and knowledge of the hardware data caching algorithms could manipulate the cache state and measure the resulting timing of an ongoing modexp operation.

However, this attack is much less feasible than the electromagnetic attack on Algorithm 1. Cache timing information will have considerable noise, and furthermore the fact that specific table entries are used is not sufficient for assembling the full exponent. Repeated measurements and considerable trial computation would typically be needed for reconstructing the full exponent.

In Algorithm 3 we again have a table which leaks cache state. Furthermore, the actual sliding operation deviates from the execution pattern k-squarings-then-a-multiplication, and as a result will leak more; an electromagnetic attack would be able to gather how many zero bits follows a k-bit block.

3.2 Leakage within arithmetic operations

We depend on multiplication, squaring, and reductions $\bmod N$ in all modexp algorithms. When we now move to specifying and analysing these operations, we need to explictly discuss computer *words*. We define a word base β and that n-bit operands have n' words in base β.

Plain quadratic ("school-book") multiplication has no data-dependent execution whatsoever, and should therefore not leak in any reasonable implementation. The same is true for special squaring code. If multiplication and squaring are done using asymptotically faster algorithms, such as Karatsuba's famous $O(n^{1.59})$ evaluate-interpolate algorithm [4] some conditional execution might sneak in. Cryptographic use of modexp requires such small operands that Karatsuba's algorithm is not beneficial at all, or at least not significantly beneficial. We will therefore assume that plain quadratic-time multiplication is used.

The reductions provide a harder problem since even the most basic division algorithm contain conditional execution; In Knuth's exposition [5] one approximates a quotient word, then applies this to an intermediate remainder, then invokes a $O(n)$ adjustment step if the approximated quotient word was too large. Even the quotient word approximation contains $O(1)$ conditional execution. The entire quotient approximation is actually conditionally executed, since it fails for certain (rare) operands. The quotient approximation might make use of hardware division instructions, which often have data-dependent timing characteristics. Schemes for avoiding hardware division [2] [8] can be of some help, if implemented for side-channel silence.

High-performance modexp implementations use Montgomery residue representation [9]. Montgomery defines a function REDC, which first performs a Hensel-norm reduction and then divides by $\beta^{n'}$, a reduction that yields an integer since the Hensel norm has created n' low zero words. In order to form a Montgomery residue, one pre-multiplies residues by $\beta^{n'} \bmod N$. In order to multiply two Montgomery residues A and B, one computes $C' = AB$ using a plain multiplication, then lets $C = REDC(C', N)$ which reduces $\bmod N$ and removes the extra factor $\beta^{n'}$.

Since REDC requires $N^{-1} \bmod \beta$ where β is the word base, it only handles odd N (at least on a binary computer).

REDC(C, N)

INPUT: $C < \beta^{2n'}, N < \beta^{n'}$, where β is the word base
OUTPUT: $C \bmod N$

1 $c = -N^{-1} \bmod \beta$
2 $R \leftarrow C$
3 **for** $i \leftarrow 0, \ldots, n' - 1$
4 $\quad q \leftarrow (R/\beta^i) \cdot c \bmod \beta$ // Compute quotient word
5 $\quad R \leftarrow R + Nq\beta^i$
6 $R \leftarrow R/\beta^{n'}$ // This division is exact
7 **if** $R \geq N$
8 $\quad R \leftarrow R - N$
9 **return** R

Algorithm 4: The REDC function. It works a *word* at a time, where β is the word base. The operands C and N have n' words. The computation of c, carry propagation hidden at line 5, and the adjustment at lines 7–8 might leak side-channel information. Note that R needs $2n' + 1$ words for accomodating the sum formed at line 5.

Reductions using Montgomery residues are easier to implement without side-channel leakage. Unlike in Euclidean division, there is nothing "approximate" about the quotient word computation, and it consist of just a plain word multiply.

The computation of $c = -N^{-1} \bmod \beta$ (line 1) is conveniently done using Hensel lifting, iterating $c_{k+1} = 2c_k - c_k^2 N \bmod 2^k$, then negating the final result $c = -c_k$ for $\beta = 2^k$. The iteration in itself does not leak side-channel information, but if the starting value c_1 is taken from a table indexed by the low few bits of N, then it indeed leaks cache information.

The operation $R \leftarrow R + Nq\beta^i$ (line 5) hides potential side-channel leakage. For each iteration of the loop, we add Nq to gradually more significant positions in R, and since R continues "to the left of" the added Nq we need to propagate any carry beyond the end of the most significant word of Nq. The propagation length is data-dependent.

In order to generate principal residues $\bmod N$ we need a final conditional subtract (lines 7–8), and like anything that is conditionally executed, this leaks side-channel information.

4 Silencing the side-channel

We above identified two classes of side-channel leaks; in this section we address them separately.

A classical approach to these problems is *blinding*, at least in the context of RSA. The idea of blinding is to first multiply a ciphertext m by r^e where r is random and e is the encryption exponent. Decryption then continues as usually, computing $r^{-1}((r^e m)^d) \pmod N$. Since the secret decryption exponent d is used, a modexp implementation that does not completely protect against leaking the exponent will still be unsafe.

4.1 Silencing macro leakage

It is not hard to silence the obvious conditional-execution induced leak of Algorithm 1. Just use Algorithm 2 for $k = 1$! The table then gets two entries, $tab[0] = 1$ and $tab[1] = m$. To make multiplication by both entries look the same, $tab[0]$ must be represented with as many bits as $tab[1]$, i.e., n bits in our setting. We make this explicit in Algorithm 5.

The cost of this remedy is that we need twice as many multiplications as in the base algorithm, translating to about 35% worse modexp performance. Furthermore, it shares the cache state leakage properties of Algorithm 2.

Silencing the cache leakage of Algorithm 2 and therefore of Algorithm 5 is somewhat harder. How can we keep using a table for saving multiplications while not making any data-dependent accesses?

MODEXP(m, d, N)

INPUT: m, $d = (d_{\ell-1}d_{\ell-2}\cdots d_1 d_0)$, N

OUTPUT: $m^d \bmod N$

1 $tab[0] \leftarrow 0\ldots01$ // 1 represented with n bits

2 $tab[1] \leftarrow m$ // m represented with n bits

3 $r \leftarrow 1$

4 **for** $j \leftarrow \ell - 1, \ldots, 0$

5 $\quad r \leftarrow r^2 \bmod N$

6 $\quad r \leftarrow r \cdot tab[d_j] \bmod N$

7 **return** r

Algorithm 5: Simple modexp, but without the obvious conditional execution. The price is about $n/2$ multiplications by unity, i.e., n-bit $0\ldots01$. It is crucial that we represent unity as an n-bit value, so that multiplication by $tab[0]$ and by $tab[1]$ behaves the same.

Our approach is to *read the entire table* before each multiplication, selecting the wanted entry using bit-wise logical masking operations. The selected entry is copied to a fixed temporary area. The operation needs to loop over all 2^k pre-computed powers of m, and decide *without conditional execution* whether it is about to handle the needed m power; if it indeed is, then it creates a word-size mask with just 1-bits, else it creates a mask with just 0-bits. An idiom like `-(index == wanted_index)` in C creates such a mask; we need to make sure the compiler does not implement this using a branch. Algorithm 7 computes this mask conservatively.

Using this mask, we combine the data from the fixed temporary area with the next pre-computed m power; a mask of 1-bits selects the tabled m power, a mask of 0-bits selects the fixed temporary area. Most of the time we copy the temporary area into itself.

It is crucial that the table selection operation is implemented as conservatively as we describe herein. It might be tempting to use hardware *conditional move* or *conditional select* instructions; this would create a data dependency graph which is a function of actual exponent data bits, and thus leak side-channel information.

Algorithm 6 details the proposed efficient and side-channel silent algorithm.

This adds $\Theta(n \cdot 2^k)$ work for each k exponent bits, or $\Theta(n^2/k \cdot 2^k)$ in total extra. Clearly, the optimal value of k as a function of the exponent size becomes smaller in Algorithm 6 than the corresponding k in Algorithm 2.

4.2 Silencing leakage within arithmetic operations

We base the reductions $\bmod N$ on Montgomery's REDC, see Algorithm 4. We identified three sources of leakage in it, we now address them in turn.

The starting value for computing c (line 1 in Algorithm 4) can come from a table, but then we must reuse the read-the-full-table trick. A better approach is to use the single-bit $c_1 = 1$, which is correct since $N^{-1} \bmod 2 = 1$ for any odd N. Such a poor initial value requires a few more iterations, but since the quantity we are computing is a single word, iterations are fast.

We can handle carry propagation of the operation $R \leftarrow R + Nq\beta^i$ (line 5 in Algorithm 4) either by propagating carry all the way to the most significant word of R, not stopping just because we are propagating 0. An alternative is to keep carry-out from line 5 operation in a separate n'-word vector, which we add to R the end. Such a vector would contain words of 0 or 1.

Making the conditional subtraction (lines 7-8 in Algorithm 4) side-channel silent is simple:

1. Perform the comparison as a full subtraction, storing the result in a temporary variable. The sign of the result is the required condition.

2. Perform the actual subtraction, now masking the subtrahend on-the-fly using the previous sign to form the mask.

MODEXP(m, d, N)

INPUT: m, $d = (d_{\ell-1}d_{\ell-2}\cdots d_1 d_0)$, N, k
OUTPUT: $m^d \bmod N$

```
1  tab[0] ← 1
2  for j ← 1, ..., 2^k − 1
3      tab[j] ← tab[j − 1] · m mod N
4  r ← 1
5  j ← ℓ − 1                                  // index to most significant exponent bit
6  while j ≥ 0
7      b ← (d_j ··· d_i) where i = max(j − k + 1, 0)
8      for s ← 1, ..., j − i + 1
9          r ← r^2 mod N
10     for i ← 0, ..., 2^k − 1          // Extract table entry in side-channel silent way
11         mask ← EQ_MAKEMASK(i, b)
12         s ← bitor(bitand(s, ¬mask), bitand((tab[i], mask))
13     r ← r · s mod N                                  // Apply extracted table entry
14     j ← i − 1
15 return r
```

Algorithm 6: The suggested modexp algorithm. It is critically important that the mask value is computed arithmetically, without branches. The line assigning s applies $mask$ over entire n-bit operands, typically applying $mask$ and $\neg mask$ on the operands word-for-word.

EQ_MAKEMASK(a, b)

INPUT: $a = (a_{\ell-1}a_{\ell-2}\cdots a_1 a_0)$, $b = (b_{\ell-1}b_{\ell-2}\cdots b_1 b_0)$,
OUTPUT: A mask $111\ldots1$ if $a = b$ else a mask $000\ldots0$

```
1  r ← 0
2  for i ← 0, ..., ℓ − 1
3      r ← r or (a_i xor b_i)
4  return (r − 1) mod β
```

Algorithm 7: Create a word-sized mask of only ones if the arguments a and b are equal, else a mask of only zeros. In practice, one could use assembly intrinsics, but avoid conditional select and conditional move, since those create a data-dependent dependency graph.

The observant reader will notice that we have not completely avoided Euclidean division, since we need one initial for recoding m as $\beta^n m \bmod N$ and another for recoding 1 as $\beta^n \bmod N$. Since only two such divisions are needed per modexp, the performance is highly non-critical. Naively iterating a side-channel hardened compare-and-subtract along the lines explained above, reducing the remainder a bit at a time, would be perfectly feasible. More sophisticated and thus faster side-channel silent Euclidean division algorithms are certainly possible.

5 Silence assumptions and claims

We have modified the standard algorithms for modexp aiming for side-channel silence. Before we analyse the results, we need to make the following assumptions about the hardware:

A1. There is no side-channel leakage from the single-word operations shifting, addition, subtraction, bit-wise logic, or comparison.

A2. There is no side-channel leakage of loaded data or stored data to and from caches or memory.

A3. There is no side-channel leakage for single-word multiply operations.

We believe A1 to be true for all processors.

It seems hard to imagine a processor where A2 is not true, except perhaps for close proximity monitoring of data buses.

A3 is not true for all processors, but it is true for all concurrent mainstream 32-bit and 64-bit processors, as far as we know. Now obsolete processors, like the UltraSPARC 2 and POWER 3 had variable-time multiply; these processors in fact leak some side-channel information with our algorithm.

Theorem 1 *Under the hardware assumptions A1-A3, the Algorithm 6 does not leak side-channel information.*

Proof: Consider two separate sets of operands, m_1, d_1, N_1 and m_2, d_2, N_2, where m_1 have the same number of bits as m_2, d_1 have the same number of bits as d_2, and N_1 have the same number of bits as N_2. We now use Algorithm 6 for computing $m_1^{d_1} \bmod N_1$ and $m_2^{d_2} \bmod N_2$. The two computations follows the exact same path through the algorithm performing the exact same sequence of operations, and making the exact same load and store operations in the same order, since neither the execution path nor the memory operations depend on the actual data bits of the operands. □

Note that we may relax the requirement that operands have the same number of bits, to that they are *encoded* with the same number of bits. I.e., we may pad operands to some specific number of bits.

6 Feasibility and implementation

For each $\lceil n/k \rceil$ multiply, we read an $(n \cdot 2^k)$-bit table. We also perform k squarings for that part of the exponentiation work. Since multiplies and squarings are super-linear operations, the large linear-time table read might not be too costly. Clearly, since the multiplication work decreases about linearly with k (in all k-ary modexp algorithms), but the table increases exponentially in k, increasing k quickly causes huge cost.

We have implemented Algorithm 6 as part of the GMP library, and tuned this implementation for good performance. GMP already had a well-tuned implementation of Algorithm 3, which has allowed us to make a level-field evaluation of the new side-channel silent algorithm.

Our implementation can be found in the GMP repository at `https://gmplib.org/repo/gmp/file/tip/mpn/generic/powm_sec.c`. That file implements a function internal to the library; the documented interface is `mpz_powm_sec`. There are intermediate versions of these functions in the GMP 5.0 release series.

6.1 Timing results and conclusions

In Table 1 we compare the performance of OpenSSL, GMP's `mpz_powm` and our side-channel silent `mpz_powm_sec` on current processors for workstations and servers, as well as a processor popular in handheld devices (ARM Cortex-A9).

We have here chosen to compute $m^d \bmod N$ to 1024-bit and 2048-bit m, d, N. These operand sizes are motivated by their importance for typical cryptographic operations. The k chosen by our implementation vary between processor and operand size; it is in fact chosen from compile-time timing runs. In the examples below, it varies between 5 and 7.

In all cases, `mpz_powm`—which is only optimised for performance—is fastest. The silent `mpz_powm_sec` is between 15% and 22% slower; it is however consistently faster than OpenSSL's (leaky!) code.

7 Conclusions and Future work

We have presented an algorithm and implementation for computing $m^d \bmod N$ where m and N are assumed to have n bits, for a general n. Our algorithm computes the result in such a way that there are no side-channel leakage, under reasonable assumptions about the underlying hardware. The algorithm has been implemented for the GMP

Phenom II 3.2 GHz	**mpz_powm**	**mpz_powm_sec**	**speed loss**	**openssl**
modexp(1024)	0.45	0.55	19%	0.67
modexp(2048)	3.01	3.61	17%	4.88
Intel SBR 3.3 GHz	**mpz_powm**	**mpz_powm_sec**	**speed loss**	**openssl**
modexp(1024)	0.59	0.71	16%	0.90
modexp(2048)	4.06	4.87	17%	6.52
ARM A9 1.0 GHz	**mpz_powm**	**mpz_powm_sec**	**speed loss**	**openssl**
modexp(1024)	5.86	7.56	22%	29.4
modexp(2048)	41.0	50.8	19%	192

Table 1: Comparison of plain GMP modexp code, side-channel silent GMP modexp code, and OpenSSL. Times are in milliseconds.

bignum library, and the performance results show that the algorithm is tolerably slower than GMP's performance-optimised modexp function.

The main slowdown with the new algorithm compared a modexp optimised for performance comes from the absence of sliding and from our reading of a table of pre-computed powers for each multiplication. While it seems difficult to use sliding without leaking side-channel information, one could reduce the table reading overhead by simultaneously handling ℓ consecutive k-bit exponent blocks, reading out ℓ table entries with one sweep through the table, storing the result into ℓ temporary buffers.

8 Acknowledgements

Johan Håstad's feedback greatly improved this paper.

References

[1] David Brumley and Dan Boneh. Remote timing attacks are practical. In *In Proceedings of the 12th USENIX Security Symposium*, pages 1–14, 2003.

[2] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the SIGPLAN PLDI'94 Conference*, June 1994.

[3] Torbjörn Granlund. GNU multiple precision arithmetic library, version 5.0, September 2013. `https://gmplib.org/`.

[4] Anatolij A. Karatsuba. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962.

[5] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1998.

[6] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113, 1996.

[7] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[8] Niels Möller and Torbjörn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, February 2011.

[9] Peter L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44:519–521, 1985.