Improving SSL Handshake Performance via Batching

Hovav Shacham
hovav@cs.stanford.edu

Dan Boneh
dabo@cs.stanford.edu

Abstract

We present an algorithmic approach for speeding up SSL's performance on a web server. Our approach improves the performance of SSL's handshake protocol by up to a factor of 2.5 for 1024-bit RSA keys. It is designed for heavily-loaded web servers handling many concurrent SSL sessions. We improve the server's performance by *batching* the SSL handshake protocol. That is, we show that b SSL handshakes can be done faster as a batch than doing the b handshakes separately one after the other. Experiments show that taking $b = 4$ leads to optimal results, namely a speedup of a factor of 2.5. Our starting point is a technique due to Fiat for batching RSA decryptions. We improve the performance of batch RSA and describe an architecture for using it in an SSL web server. We give experimental results for all the proposed techniques.

1 Introduction

The Secure Socket Layer (SSL) is the most widely deployed protocol for securing communication on the World Wide Web (WWW). The protocol is used by most e-commerce and financial web sites. It guarantees privacy and authenticity of information exchanged between a web server and a web browser. Unfortunately, SSL is not cheap. A number of studies show that web servers using the SSL protocol perform far worse than web servers who do not secure web traffic. This forces web sites using SSL to buy significantly more hardware in order to provide reasonable response times.

Here we propose a software-only approach for speeding up SSL: batching the SSL handshakes on the web server. The basic idea is as follows: the web server waits until it receives b handshake requests from b different clients. It then treats these b handshakes as a batch and performs the necessary computations for all b handshakes at once. Our experiments show that, for $b = 4$, batching the SSL handshakes in this way results in a factor of 2.5 speedup over doing the b handshakes sequentially, without requiring any additional hardware.

Our starting-point is a technique due to Fiat [5] for batch RSA decryption. Fiat suggested that one can decrypt multiple RSA ciphertexts as a batch faster than decrypting them one by one. Unfortunately, our experiments show that Fiat's basic algorithm, naively implemented, does not give much improvement for key sizes commonly used in SSL handshakes. Our first set of results, given in Section 3, shows how to batch RSA decryption in a way that gives a significant speedup with common RSA keys.

In Section 4 we present an architecture for a batching web server and discuss several scheduling issues. As we will see, a batching web server must manage multiple public key certificates. Consequently, a batching web server must employ a scheduling algorithm that assigns certificates to incoming connections, and picks batches from pending requests, so as to optimize server performance.

Finally, in Section 5 we describe our experiments and give running times for various key sizes and various loads on the web server.

1.1 Preliminaries

As discussed above, this paper focuses on improving the performance of the SSL handshake protocol. The handshake protocol is part of the bottleneck that significantly degrades server performance.

SSL Handshake. For completeness we briefly describe the SSL handshake protocol. We note that SSL supports several handshake mechanisms. The one described below is the simplest and is the most commonly used. More information can be found in [4].
Step 1: the web browser connects to the web server and sends a client-hello.
Step 2: the web server responds with a server-hello message sequence. These messages contain the server's certificate, which in turn contains the server's RSA public key.
Step 3: The browser picks a random 48-byte string R and encrypts it using the web server's public RSA key. Let C be the resulting ciphertext. The web browser sends a client-key-exchange message which contains C. The string R is called the pre-master-secret.
Step 4: The web server obtains the pre-master-secret R by using its private RSA key to decrypt C. Both the browser and server then derive the session keys from R and some other shared information.

RSA Public Keys. Step 4 above is the expensive step in the SSL handshake since it requires the server to perform an RSA decryption. To describe our speedup of the SSL handshake we must first briefly review the RSA cryptosystem [9]. We refer to [8] for a complete description.

An RSA public key is made of two integers $\langle N, e\rangle$. Here $N = pq$ is the product of two large primes, and is typically 1024 bits long. The value e is called the encryption exponent and is typically some small number such as $e = 3$ or $e = 65537$. Both N and e are embedded in the server's public-key certificate. The RSA private key is an integer d satisfying $e \cdot d = 1 \bmod (p-1)(q-1)$.

To encrypt a message M using an RSA public key $\langle N, e\rangle$, one first formats the message M to obtain an integer X in $\{1, \ldots, N\}$. This formatting is often done using the PKCS1 standard [1, 7]. The ciphertext is then computed as $C = X^e \bmod N$. Recall that the web browser does this in Step 3 of the SSL handshake protocol.

To decrypt a ciphertext C the web server uses its private key d to compute the e'th root of C in $\mathbb{Z}_N$. The eth root of C is given by $C^d \bmod N$. Since both d and N are large numbers (each 1024 bits long) this is a lengthy computation on the web server. We note that d must be taken as a large number (i.e., on the order of N) since otherwise the RSA system is insecure [2, 11].

2 Review of Fiat's Batch RSA

Fiat [5] is the first to propose speeding up RSA decryption via batching. We briefly review Fiat's proposal and describe our improvements in the next section. For the rest of the paper all arithmetic is done modulo N, except where otherwise noted.

Fiat observed that when using small public exponents e_1 and e_2 it is possible to decrypt two ciphertexts for approximately the price of one. Suppose v_1 is a ciphertext obtained by encrypting using the public key $\langle N, 3\rangle$. Similarly, v_2 is a ciphertext obtained by encrypting using the public key $\langle N, 5\rangle$. To decrypt v_1 and v_2 we must compute $v_1^{1/3}$ and $v_2^{1/5} \bmod N$. Fiat observed that by setting $A = (v_1^5 \cdot v_2^3)^{1/15}$ we obtain

$$v_1^{1/3} = \frac{A^{10}}{v_1^3 \cdot v_2^2} \qquad \text{and} \qquad v_2^{1/5} = \frac{A^6}{v_1^2 \cdot v_2}$$

Hence, at the cost of computing a single 15'th root we are able to decrypt both v_1 and v_2. Note that some extra arithmetic is required.

This batching technique is only worthwhile when the public exponents e_1 and e_2 are very small (e.g., 3 and 5). Otherwise, the extra arithmetic required is too expensive. Also, notice that one can only batch-decrypt ciphertexts encrypted using *distinct public exponents*. This is essential. Indeed, in Appendix A we show (using simple Galois theory) that it is not possible to batch when the same public key is used. That is, it is not possible to batch the computation of $v_1^{1/3}$ and $v_2^{1/3}$.

Fiat generalized the above observation to the decryption of a batch of b RSA ciphertexts. We have b distinct and pairwise relatively prime public keys $e_1, \ldots, e_b$, all sharing a common modulus $N = pq$. Furthermore, we have b encrypted messages $v_1, \ldots, v_b$, one encrypted with each key, which we wish to decrypt simultaneously, to obtain the plaintexts $m_i = v_i^{1/e_i}$.

The batch process is implemented around a complete binary tree with b leaves, with the additional property that every inner node has two children. Our notation will be biased towards expressing locally recursive algorithms: Values will be percolated up and down the tree. With one exception noted later, quantities subscripted by L or R refer to the corresponding value of the left or right child of the node, respectively. For example, m is the value of m at a node; m_{R} is the value of m at that node's right child.

Some values necessary to batching depend only on the particular placement of keys in the tree, and may be precomputed and reused for multiple batches. We will denote precomputed values in the batch tree with capital letters, and values that are computed in a particular decryption with lower-case letters.

Fiat's algorithm consists of three phases: an upward-percolation phase, an exponentiation phase, and a downward-percolation phase. We consider each in turn.

Upward-percolation. In the upward-percolation phase, we seek to combine the individual encrypted messages v_i to form, at the root of the batch tree, the value $v = \prod_{i=1}^{b} v_i^{e/e_i}$, where $e = \prod_{i=1}^{b} e_i$.

In preparation, we assign to each leaf node a public exponent: $E \leftarrow e_i$. Each inner node then has its E computed as the product of those of its children: $E \leftarrow E_{\mathrm{L}} \cdot E_{\mathrm{R}}$. (The root node's E will be equal to e, the product of all the public exponents.)

Each encrypted message v_i is placed (as v) in the leaf node labeled with its corresponding e_i. The v's are percolated up the tree using the following recursive step, applied at each inner node:

$$v \leftarrow v_{\mathrm{L}}^{E_{\mathrm{R}}} \cdot v_{\mathrm{R}}^{E_{\mathrm{L}}}. \tag{1}$$

Exponentiation-phase. At the completion of the upward-percolation phase, the root node contains $v = \prod_{i=1}^{b} v_i^{e/e_i}$. In the exponentiation phase, the eth root of this v is extracted. (In the basic Fiat scheme, this is the only point at which knowledge of the factorization of N is required.) The exponentiation yields $v^{1/e} = \prod_{i=1}^{b} v_i^{1/e_i}$, which we store as m in the root node.

Downward-percolation. In the downward-percolation phase, we seek to break up the product m into its constituent subproducts m_{L} and m_{R}, and, eventually, into the decrypted messages m_i at the leaves.

Fiat gives a method for accomplishing this breakup. At each inner node we choose an X satisfying the following simultaneous congruences:

$$X = 0 \pmod{E_{\mathrm{L}}} \qquad\qquad X = 1 \pmod{E_{\mathrm{R}}}$$

We construct X using the Chinese Remainder Theorem. Two further numbers, X_{L} and X_{R}, are defined at each node as follows:

$$X_{\mathrm{L}} = X/E_{\mathrm{L}} \qquad\qquad X_{\mathrm{R}} = (X-1)/E_{\mathrm{R}}$$

Both divisions are done over the integers. (There is a slight infelicity in the naming here: X_{L} and X_{R} are not the same as the X's of the node's left and right children, as implied by the use of the L and R subscripts, but separate values.)

As Fiat shows, X, X_{L}, and X_{R} are such that, at each inner node, m^X equals $v_{\mathrm{L}}^{X_{\mathrm{L}}} \cdot v_{\mathrm{R}}^{X_{\mathrm{R}}} \cdot m_{\mathrm{R}}$. This immediately suggests the recursive step used in downward-percolation:

$$m_{\mathrm{R}} \leftarrow m^X / \left(v_{\mathrm{L}}^{X_{\mathrm{L}}} \cdot v_{\mathrm{R}}^{X_{\mathrm{R}}}\right) \qquad\qquad m_{\mathrm{L}} \leftarrow m/m_{\mathrm{R}} \tag{2}$$

At the end of the downward-percolation process, each leaf's m contains the decryption of the v placed there originally. Only one large (full-size) exponentiation is needed, instead of b of them. In addition, the process requires a total of 4 small exponentiations, 2 inversions, and 4 multiplications at each of the $b-1$ inner nodes.

3 Improved Batching

Basic batch RSA is fast with very large moduli, but not a big improvement with moderate-size moduli. This is because batching is essentially a tradeoff: more auxiliary operations in exchange for fewer full-strength exponentiations.

Since we are experimenting with batching in an SSL-enabled web server we must focus on key sizes generally employed on the web, e.g., $n = 1024$ bits. We also limit the batch size b to small numbers, on the order of $b = 4$, since collecting large batches can introduce unacceptable delay. For simplicity of analysis (and implementation), we further restrict our attention to values of b that are powers of 2.

In this section we describe a number of improvements to batch RSA that lead to a significant speedup in a batching web server.

3.1 Division Speedups

Fiat's scheme presented in the previous section performs two divisions at each internal node, for a total of $2b - 2$ required modular inversions. Modular inversions are asymptotically faster than large modular exponentiations [6]. In practice, however, modular inversions are costly. Indeed, our first implementation (with $b = 4$ and a 1024-bit modulus) spent more time doing the inversions than doing the large exponentiation at the root.

We present two techniques that, when combined, require only a single modular inversion throughout the algorithm. The cost is an additional $O(b)$ modular multiplications. This tradeoff gives a substantial running-time improvement.

3.1.1 Delayed Division.

An important realization about the downward-percolation phase given in Equation (2) is that the actual value of m for the internal nodes of the tree is consulted only for calculating m_{L} and m_{R}. An alternative representation of m that allows the calculation of m_{L} and m_{R} and can be evaluated at the leaves to yield m would do just as well.

We convert a modular division a/b to a "promise," $\langle a, b\rangle$. We can operate on this promise as though it were a number, and, when we need to know its value, we can "force" it by actually computing $b^{-1}a$.

Operations on these promises work in the obvious way (similar to operations in projective coordinates):

$$
\begin{aligned}
a/b &= \langle a, b\rangle & \qquad \langle a, b\rangle^c &= \langle a^c, b^c\rangle \\
c \cdot \langle a, b\rangle &= \langle ac, b\rangle & \qquad \langle a, b\rangle \cdot \langle c, d\rangle &= \langle ac, bd\rangle \\
\langle a, b\rangle / c &= \langle a, bc\rangle & \qquad \langle a, b\rangle \,/\, \langle c, d\rangle &= \langle ad, bc\rangle
\end{aligned}
$$

Multiplications and exponentiations take twice as much work as otherwise, but division can be computed without resort to modular inversion.

If, after the exponentiation at the root, we express the root m as a promise, $\mathbf{m} \leftarrow \langle m, 1\rangle$, we can easily convert the downward-percolation step in (2) to employ promises:

$$
\mathbf{m}_{\mathrm{R}} \leftarrow \mathbf{m}^X \big/ \left(v_{\mathrm{L}}^{X_{\mathrm{L}}} \cdot v_{\mathrm{R}}^{X_{\mathrm{R}}}\right) \qquad\qquad \mathbf{m}_{\mathrm{L}} \leftarrow \mathbf{m}/\mathbf{m}_{\mathrm{R}} \tag{3}
$$

No internal inversions are required. The promises can be evaluated at the leaves to yield the decrypted messages.

Batching using promises requires $b - 1$ additional small exponentiations and $b - 1$ additional multiplications, one each at every inner node, and saves $2(b-1) - b = b - 2$ inversions.

3.1.2 Batched Division.

To reduce further the number of inversions, we use batched divisions. When using delayed inversions (as described in the previous section) one division is needed for every leaf of the batch tree. We show that these b divisions can be done at the cost of a *single* inversion with a few more multiplications.

Suppose we wish to invert three values x, y, and z. We can proceed as follows: we form the partial products yz, xz, and xy; and we form the total product xyz and invert it, yielding $(xyz)^{-1}$. With these values, we can calculate all the inverses:

$$x^{-1} = (yz) \cdot (xyz)^{-1} \qquad\qquad y^{-1} = (xz) \cdot (xyz)^{-1} \qquad\qquad z^{-1} = (xy) \cdot (xyz)^{-1}$$

Thus we have obtained the inverses of all three numbers, at the cost of only a single modular inverse along with a number of multiplications. More generally, we obtain the following lemma:

Lemma 3.1. *Let $x_1, \ldots, x_n \in \mathbb{Z}_N$. Then all n inverses $x_1^{-1}, \ldots, x_n^{-1}$ can be obtained at the cost of one inversion and $3n - 3$ multiplications.*

Proof. A general batched-inversion algorithm proceeds, in three phases, as follows. First, set $A_1 \leftarrow x_1$, and $A_i \leftarrow x_i \cdot A_{i-1}$ for $i > 1$. It is easy to see, by induction, that

$$A_i = \prod_{j=1}^{i} x_j \ . \tag{4}$$

Next, invert $A_n = \prod x_j$, and store the result in B_n: $B_n \leftarrow (A_n)^{-1} = \prod x_j^{-1}$. Now, set $B_i \leftarrow x_{i+1} \cdot B_{i+1}$ for $i < n$. Again, it is easy to see that

$$B_i = \prod_{j=1}^{i} x_j^{-1} \ . \tag{5}$$

Finally, set $C_1 \leftarrow B_1$, and $C_i \leftarrow A_{i-1} \cdot B_i$ for $i > 1$. We have $C_1 = B_1 = x_1^{-1}$, and, combining (4) and (5), $C_i = A_{i-1} \cdot B_i = x_i^{-1}$ for $i > 1$. We have thus inverted each x_i.

Each phase above requires $n - 1$ multiplications, since one of the n values is available without recourse to multiplication in each phase. Therefore, the entire algorithm computes the inverses of all the inputs in $3n - 3$ multiplications and a single inversion. □

Batched division can be combined with delayed division: The promises at the leaves of the batch tree are evaluated using batched division. Consequently, only a single modular inversion is required for the entire batching procedure. We note that the batch division algorithm of Lemma 3.1 can be easily modified to conserve memory and store only n intermediate values at any given time.

3.2 Global Chinese Remainder

It is standard practice to employ the Chinese Remainder Theorem (CRT) in calculating RSA decryptions. Rather than compute $m \leftarrow v^d \pmod{N}$, one evaluates modulo p and q:

$$m_p \leftarrow v_p^{d_p} \pmod{p} \qquad\qquad m_q \leftarrow v_q^{d_q} \pmod{q}$$

Here $d_p = d \bmod p - 1$ and $d_q = d \bmod q - 1$. Then one uses the CRT [6] to calculate m from m_p and m_q. This is approximately 4 times faster than evaluating m directly [8].

This idea extends naturally to batch decryption. We reduce each encrypted message v_i modulo p and q. Then, instead of using a single batch tree modulo N, we use two separate, parallel batch trees, modulo p and q, and then combine the final answers from both using the CRT. Batching in each tree takes between a quarter and an eighth as long as in the original, unified tree (since the number-theoretical primitives employed, as commonly implemented, take quadratic or cubic time in the bit-length of the modulus), and the b CRT steps required to calculate each m_i mod N afterwards take negligible time compared to the accrued savings.

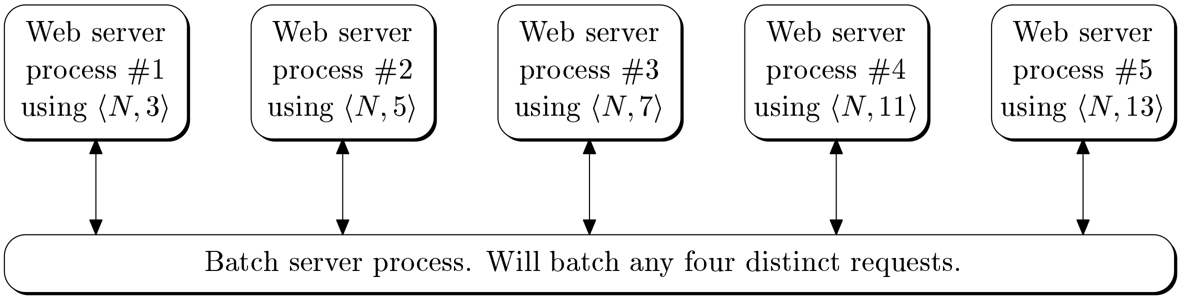

Figure 1: A batching web server using a 4-of-5 batching architecture

3.3 Simultaneous Multiple Exponentiation

Simultaneous multiple exponentiation (see [8], §14.6) provides a method for calculating $a^u \cdot b^v \bmod m$ without first evaluating a^u and b^v. It requires approximately as many multiplications as does a single exponentiation with the larger of u or v as exponent.

For example, in the percolate-upward step, $V \leftarrow V_{\mathrm{L}}^{E_{\mathrm{R}}} \cdot V_{\mathrm{R}}^{E_{\mathrm{L}}}$, the entire right-hand side can be computed in a single multiexponentiation. The percolate-downward step involves the calculation of the quantity $V_{\mathrm{L}}^{X_{\mathrm{L}}} \cdot V_{\mathrm{R}}^{X_{\mathrm{R}}}$, which can be accelerated similarly.

These small-exponentiations-and-product calculations are a large part of the extra bookkeeping work required for batching. Using Simultaneous multiple exponentiation cuts the time required to perform them by close to 50%.

3.4 Node Reordering

There are two factors that determine performance for a particular batch of keys.

First, smaller encryption exponents are better. The number of multiplications required for evaluating a small exponentiation is proportional to the number of bits in the exponent. Since upward and downward percolation both require $O(b)$ small exponentiations, increasing the value of $e = \prod e_i$ can have a drastic effect on the efficiency of batching.

Second, some exponents work well together. In particular, the number of multiplications required for a simultaneous multiple exponentiation is proportional to the number of bits in the larger of the two exponents. If we can build batch trees that have balanced exponents for multiple exponentiation (E_{L} and E_{R}, then X_{L} and X_{R}, at each inner node), we can streamline the multi-exponentiation phases.

With $b = 4$, optimal reordering is fairly simple. Given public exponents $e_1 < e_2 < e_3 < e_4$, the arrangement e_1–e_4–e_2–e_3 minimizes the disparity between the exponents used in simultaneous multiple exponentiation in both upward and downward percolation. Rearranging is harder for $b > 4$.

4 Architecture for a Batching Web Server

Building the batch RSA algorithm into real-world systems presents a number of architectural challenges. Batching, by its very nature, requires an aggregation of requests. Unfortunately, commonly-deployed protocols and programs were not designed with RSA aggregation in mind. Our solution is to create a batching server process that provides its clients with a decryption oracle, abstracting away the details of the batching procedure.

With this approach we minimize the modifications required to the existing servers. Moreover, we simplify the architecture of the batch server itself by freeing it from the vagaries of the SSL protocol. An example of the resulting web server design is shown in Figure 1. Note that batching requires that the web server manage multiple certificates, i.e., multiple public keys, all sharing a common modulus N. We describe the various issues with this design in the subsections below.

4.1 The Two-Tier Model

For a protocol that calls for public-key decryption, the presence of a batch-decryption server induces a two-tier model. First is the batch server process, which aggregates and performs RSA decryptions. Next are client processes that send decryption requests to the batch server. These client processes implement the higher-level application protocol (e.g., SSL) and interact with end-user agents (e.g., browsers).

Hiding the workings of the decryption server from its clients means that adding support for batch RSA decryption to existing servers (such as ApacheSSL) engenders roughly the same changes as adding support for hardware-accelerated decryption. The only additional challenge is in assigning the different public keys to the end-users; here the hope is to obtain roughly equal numbers of decryption requests with each e_i. End-user response times are highly unpredictable, so there is a limit to the cleverness that may be usefully employed in the public key distribution.

One solution that seems to work: If there are k keys (each with a corresponding certificate), spawn ck web server processes, and assign to each a particular key. This approach has the advantage that individual server processes need not be aware of the existence of multiple keys. The correct value for c depends on factors such as the load on the site, the rate at which the batch server can perform decryption, and the latency of the communication with the clients.

We discuss additional ways of accommodating workload unpredictability in the next subsection.

4.2 Decryption Server Scheduling

The batch server performs a set of related tasks. It receives requests for decryption, each of which is encrypted with a particular public exponent e_i; it aggregates these into batches as well as it can; it performs the batch decryption described in Section 3, above; finally, it responds to the requests with the corresponding plaintexts.

The first and last of these tasks are relatively simple I/O problems; the decryption stage has already been discussed. What remains is the scheduling step: Of the outstanding requests, which should we batch? This question gives rise to a related one: If no batch is available, what action should we take?

We designed our batching server with three scheduling criteria: maximum throughput, minimum turnaround time, and minimum turnaround-time variance. The first two criteria are self-evident; the third may require some motivation. Lower turnaround-time variance means the server's behavior is more consistent and predictable, and helps prevent client timeouts. It also tends to prevent starvation of requests, which is a danger under more exotic scheduling policies.

Under these constraints, a batch server's scheduling should implement a queue, in which older requests are handled first, if possible. At each step, the server seeks the batch that allows it to service the oldest outstanding requests.

We cannot compute a batch that includes more than one request encrypted with any particular public exponent e_i. This immediately leads to the central realization about batch scheduling: It makes no sense, in a batch, to service a request that is not the oldest for a particular e_i; substituting the oldest request for a key into the batch improves the overall turnaround-time variance and makes the batch server better approximate a perfect queue.

Therefore, in choosing a batch, we need only consider the oldest pending request for each e_i. To facilitate this, the batch server keeps k queues Q_i, one for each key. When a request arrives, it is enqueued onto the queue that corresponds to the key with which it was encrypted; this takes $O(1)$ time. In choosing a batch, the server examines only the heads of each of the queues.

Suppose that there are k keys, with public exponents $e_1, \ldots, e_k$, and that the server decrypts requests in batches of b messages each. (We will see a reason why we might want to choose k larger than b in Section 4.4, below.) The correct requests to batch are the b oldest requests from amongst the k queue heads. If we keep the request queues Q_i in a heap (see, for example, [3]), with priority determined by the age of the request at the queue head, then batch selection can be accomplished thus: extract the maximum (oldest-head) queue from the heap; dequeue the request at its head, and repeat to obtain b requests to batch. After the batch has been selected, the b queues from which requests were taken may be replaced in the heap. The entire process takes $O(b \lg k)$ time.

```
Batch-Server()
while true
do Request-Wait-With-Timeout()
     if Requests-Arrived()
       then Enqueue-Requests()
            b ← Pick-Batch()
            if b ≠ nil
              then Do-Batch(b)
       else b ← Pick-Batch()
            if b ≠ nil
              then Do-Batch(b)
              else r ← Pick-Single()
                   if r ≠ nil
                     then Do-Single(r)
```

Figure 2: Batch server scheduling algorithm

4.3 Multi-Batch Scheduling

Note that the process described above picks only a single batch to perform. It would be possible to attempt to choose several batches at once; this would allow more batching in some cases. For example, with $b = 2$, $k = 3$, and requests for the keys 3–3–5–7 in the queues, the one-step lookahead may choose to do a 5–7 batch first, after which only the unbatchable 3–3 remain. A smarter server could choose to do 3–5 and 3–7 instead.

The algorithms for doing lookahead are somewhat messier than the single-batch ones. Additionally, since they take into account factors other than request age, they can worsen turnaround-time variance or lead to request starvation.

There is a more fundamental objection to multi-batch lookahead. Performing a batch decryption takes a significant amount of time; accordingly, if the batch server is under load, additional requests will have arrived by the time the first chosen batch has been completed. These may make a better batch available than was without the new requests. (If the batch server is not heavily loaded, batching is not important, as explained in Section 4.4, below.)

4.4 Server-Load Considerations

Not all servers are always under maximal load. Server design must take different load conditions into account.

Our server reduces latency in a medium-load environment as follows: we use k public keys on the web server and allow batching of any subset of b of them, for some $b < k$. This has some costs: we must pre-construct and keep in memory the constants associated with $\binom{k}{b}$ batch trees, one for each set of e's.

However, we need no longer wait for exactly one request with each e before a batch is possible. For k keys batched b at a time, the expected number of requests required to give a batch is

$$E[\#\text{ requests}] = k \cdot \sum_{i=1}^{b} \frac{1}{k - i + 1}. \qquad (6)$$

Here we are assuming each incoming request uses one of the k keys randomly and independently. With $b = 4$, moving from $k = 4$ to $k = 6$ drops the expected length of the request queue at which a batch is available by more than 31%, from 8.33 to 5.70.

The particular relationship of b and k can be tuned for a particular server. The batch-selection algorithm described in Section 4.2, above, has time-performance logarithmic in k, so the limiting factor on k is the size of the kth prime, since particularly large values of e degrade the performance of batching.

In low-load situations, requests trickle in slowly, and waiting for a batch to be available may introduce unacceptable latency. A batch server must have some way of falling back on unbatched RSA decryption.

batch size	key size 512	768	1024	1536	2048
(unbatched)	1.53	4.67	8.38	26.10	52.96
2	1.22	3.09	5.27	15.02	29.43
4	0.81	1.93	3.18	8.63	16.41
8	0.70	1.55	2.42	6.03	10.81

Table 1: RSA decryption time, in milliseconds, as a function of batch and key size

Conversely, if a batch is available, batching is a better use of processor time than unbatched RSA. So, by the considerations given in Section 4.3, above, the batch server should perform only a single unbatched decryption, then look for new batching opportunities.

Scheduling the unbatched decryptions introduces some complications. The obvious algorithm — when requests arrive, do a batch if possible, otherwise do a single unbatched decryption — leads to undesirable real-world behavior. The batch server tends to exhaust its queue quickly. Then it responds immediately to each new request, and so never accumulates enough requests to batch.

We chose a different approach, which does not exhibit the performance degeneration described above. The server waits for new requests to arrive, with a timeout. When new requests arrive, it adds them to its queues. If a batch is available, it evaluates it. The server falls back on unbatched RSA decryptions only when the request-wait times out. This approach increases the server's turnaround-time under light load, but scales gracefully in heavy use. The timeout value is, of course, tunable.

The server's scheduling algorithm is given in Fig. 2.

5 Performance

We measured the performance of the batch RSA decryption method described in Section 3, and of the batch server described in Section 4. These tests show a marked improvement over unbatched RSA and SSL at standard key sizes.

Timing was performed on a machine with an Intel Pentium III processor clocked at 750 MHz and 256 MB RAM. For SSL handshake measurements the client machine (used to drive the web server) featured dual Intel Pentium III processors clocked at 700 MHz and 256 MB RAM. The two machines were connected via switched fast Ethernet. The underlying cryptography and SSL toolkit was OpenSSL 0.9.5.

5.1 RSA Decryption

Since modular exponentiation is asymptotically more expensive than the other operations involved in batching, the gain from batching approaches a factor-of-b improvement only when the key size is improbably large. With 1024-bit RSA keys the overhead is relatively high, and a naive implementation is slower than unbatched RSA. The improvements described in Section 3 are intended to lower the overhead and improve performance with small batches and standard key-sizes. The results are described in Table 1. In all experiments we used the smallest possible values for the encryption exponent e.

Batching provides almost a factor-of-five improvement over plain RSA with $b = 8$ and $n = 2048$. This is to be expected. More important, even with standard 1024-bit keys, batching improves performance significantly. With $b = 4$, RSA decryption is accelerated by a factor of 2.6; with $b = 8$, by a factor of almost 3.5. These improvements can be leveraged to improve SSL handshake performance.

At small key sizes, for example $n = 512$, an increase in batch size beyond $b = 4$ provides only a modest improvement in RSA performance. Because of the increased latency that large batch sizes impose on SSL handshakes, especially when the web server is not under high load, large batch sizes are of limited utility for real-world deployment.

batch size	load 16	32	48
(unbatched)	105	98	98
2-of-2	149	141	134
4-of-4	218	201	187
4-of-6	215	198	185
8-of-8	274	248	227

Table 2: SSL handshakes per second as a function of batch size. 1024 bit keys.

5.2 SSL Handshake

To measure SSL handshake performance improvements using batching, we wrote a simple web server that responds to SSL handshake requests and simple HTTP requests. The server uses the batching architecture described in Section 4. The web server is a pre-forked server, relying on "thundering herd" behavior for scheduling [10, §27.6]. All pre-forked server processes contact an additional batching server process for all RSA decryptions, as described in Section 4.

Our multi-threaded SSL test client bombards the web server with concurrent `HTTP HEAD` requests. The server sends a 187-byte response. Handshake throughput results for 1024-bit RSA keys are summarized in Table 2, above. Here "load" is the number of simultaneous connections the client makes to the server. The "b-of-k" in the first column refers to a total of k distinct public exponents on the server where any subset of b can be batched. See Section 4.4.

The tests above measure server performance under a constant high load, so moving from $k = 4$ to $k = 6$ provides no advantage.

Batching is clearly an improvement, increasing handshake throughput by a factor of 2.0 to 2.5, depending on the batch size. At better than 200 handshakes per second, the batching web server is competitive with hardware-accelerated SSL web servers, without the need for expensive specialized hardware.

6 The Downside of Batch SSL

As we saw in previous sections, batching SSL handshakes leads to a significant improvement on the web server. Nevertheless, there are a few issues with using the batching technique. Below, we discuss these issues, by order of severity.

- When using batching, the web-server administrator must obtain multiple certificates for the web site. In the previous section we gave the example of obtaining four or six certificates (all using the same RSA modulus). We note that these certificates are used by the same site and consequently have the same X.500 Distinguished Name. In an ideal world, Certificate Authorities (CA's) would issue multiple certificates (using a single RSA modulus) for the same site at no extra charge. Unfortunately, currently CA's charge per certificate regardless of whether the certificate is for the same site.

- Batching relies on RSA with very small public exponents, namely $e = 3, 5, 7, 11$, etc. Although there are no known attacks on the resulting handshake protocol, web sites commonly use a slightly larger public exponent, namely $e = 65537$. This is not a serious concern, but is worth mentioning.

- One might wish to further speed up batching by using a commercial off-the-shelf crypto hardware accelerator. This works fine — the accelerator can be used to perform the full RSA decryption at the top of the batching tree. However, the main CPU has to perform all the other computations involved in batching. The main CPU has to percolate values up the tree and back down the tree. Consequently, when using batching, the CPU has to work harder per handshake, compared to regular RSA, where the entire decryption is done on the card. Hence, although handshake time is reduced, the CPU has less time for other web tasks. Ideally, one would expect the accelerator card to perform the entire batching process.

7 Conclusions

We presented the first implementation of batch RSA in an SSL web server. Our first set of results describes several substantial improvements to the basic batch RSA decryption algorithm. We showed how to reduce the number of inversions in the batch tree to a single inversion. We obtained a further speedup by proper use of the CRT and use of simultaneous multiple exponentiation.

We also presented an architecture for building a batching SSL web server. The architecture is based on using a batch server process that functions as a fast decryption oracle for the main web server processes. The batching server process includes a scheduling algorithm to determine which subset of pending requests to batch.

Our experiments show a substantial speedup to the SSL handshake. We hope these results will promote the use of batching to speed up secure web servers. We intend to make our code available for anyone wishing to experiment with it.

Acknowledgments

We thank Nick Howgrave-Graham for improving our batch division algorithm.

References

[1] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Proceedings of Eurocrypt '94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, 1994.

[2] D. Boneh and G. Durfee. Cryptanalysis of RSA with private key d less than $n^{0.292}$. In *Proceedings of Eurocrypt '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 1999.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[4] T. Dierks and C. Allen. RFC 2246: The TLS Protocol Version 1, January 1999.

[5] A. Fiat. Batch RSA. In *Proceedings of Crypto '89*, pages 175–185, 1989.

[6] Donald Ervin Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms.* Addison-Wesley, 3rd edition, 1998.

[7] RSA Labs. Public Key Cryptography Standards (PKCS), number 1.

[8] A. J. (Alfred J.) Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* The CRC Press series on discrete mathematics and its applications. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1997.

[9] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Commun. of the ACM*, 21:120–126, 1978.

[10] W. Richard Stevens. *UNIX Network Programming*, volume 1. Prentice Hall PTR, second edition, 1998.

[11] M. Wiener. Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Info. Th.*, 36(3):553–558, 1990.

Appendix A: Impossibility of Batching with a Single Public Key

Fiat showed that when using relatively prime public exponents e_1, e_2, with a common modulus, it is possible to batch the decryption of v_1, v_2. The fact that batching only works when different public exponents are used forces batching web servers to manage multiple certificates. It is natural to ask whether one can batch the decryption of two ciphertexts encrypted *using the same RSA public key*. More precisely, can we batch the computation of $v_1^{1/e}$ and $v_2^{1/e}$? We show that batching using a single public key is not possible using arithmetic operations.

Given an RSA public key $\langle N, e\rangle$ we say that batch decryption of ciphertexts v_1, v_2 is possible if there exist rational functions f, g_1, g_2 over $\mathbb{Z}_N$ and an integer m such that

$$v_1^{1/e} = g_1(A, v_1, v_2) \ ; \qquad v_2^{1/e} = g_2(A, v_1, v_2) \qquad \text{where} \qquad A = \left[f(v_1, v_2)\right]^{1/m}$$

For efficiency one would like the functions f, g_1, g_2 to be of low degree. Fiat gives such f, g_1, g_2 when relatively prime exponents e_1, e_2 are used. Fiat uses $m = e_1 \cdot e_2$. Note that batch RSA works in any field — there is nothing specific to $\mathbb{Z}_N$.

We show that no such f, g_1, g_2, m exist when a single public key is used. More precisely, we show that no such expressions exists when all arithmetic is done in characteristic 0 (e.g., over the rationals). Since batching is generally oblivious to the underlying field, our inability to batch in characteristic 0 indicates that no such batching exists in $\mathbb{Z}_N$ either.

Let $\mathbb{Q}$ be the field of rational numbers, and $v_1, v_2 \in \mathbb{Q}$. The existence of g_1, g_2 implies that $\mathbb{Q}[v_1^{1/e}, v_2^{1/e}]$ is a subfield of $\mathbb{Q}[A]$ for all v_1, v_2. This cannot be, as stated in the following lemma:

Lemma 7.1. *For any $e > 1$ and f, g_1, g_2, m as above, there exist $v_1, v_2 \in \mathbb{Q}$ such that $\mathbb{Q}[v_1^{1/e}, v_2^{1/e}]$ is not a subfield of $\mathbb{Q}[f(v_1, v_2)^{1/m}]$*

Proof Sketch Let f, g_1, g_2, m be a candidate batching scheme. Let v_1, v_2 be distinct integer primes and set $A = f(v_1, v_2)$. We show that $\mathbb{Q}[v_1^{1/e}, v_2^{1/e}]$ is not a subfield of $\mathbb{Q}[A^{1/m}]$. Consequently, f, g_1, g_2, m is an invalid batching scheme.

Let $K = \mathbb{Q}[v_1^{1/e}, v_2^{1/e}]$ and $L = \mathbb{Q}[A^{1/m}]$. We know that $[K : \mathbb{Q}] = e^2$. Similarly $[L : \mathbb{Q}] = m'$ for some m' dividing m. Assume, towards a contradiction, that K is a subfield of L. Then $[K : \mathbb{Q}]$ divides $[L : \mathbb{Q}]$. Hence, e divides m.

Define L_0 as an extension of L by adjoining a primitive m'th root of unity. Then L_0 is a Galois extension of $\mathbb{Q}$. Similarly, let K_0 be an extension of K by adjoining a primitive m'th root of unity. Then K_0 is a Galois extension of $\mathbb{Q}$ (since by assumption e divides m). Observe that if $K \subseteq L$ then $K_0 \subseteq L_0$. Consequently, to prove the lemma it suffices to show that $K_0 \not\subseteq L_0$.

Let T be an extension of $\mathbb{Q}$ obtained by adjoining a primitive m'th root of unity. Then K_0 and L_0 are Galois extensions of T. To show that K_0 is not contained in L_0 we consider the Galois group of K_0 and L_0 over T.

Let G be the Galois group of K_0 over T and let H be the Galois group of L_0 over T. If $K_0 \subseteq L_0$ then the fundamental theorem of Galois theory says that there exists a normal subgroup H_0 of H such that $G = H/H_0$. Hence, it suffices to prove that G does not arise as a factor group of H.
The lemma now follows from the following three simple facts:

Fact 1: The Galois group G is isomorphic to $\mathbb{Z}_e \times \mathbb{Z}_e$.
Fact 2: The Galois group H is isomorphic to $\mathbb{Z}_{m'}$.
Fact 3: For any pair m', e the group $\mathbb{Z}_{m'}$ does not have a factor group isomorphic to $\mathbb{Z}_e \times \mathbb{Z}_e$.

Fact 3 follows since all factor groups of $\mathbb{Z}_{m'}$ are cyclic, but $\mathbb{Z}_e \times \mathbb{Z}_e$ is not. $\square$

To conclude, we note that the proof shows that any batching scheme f, g_1, g_2, m will fail to work correctly in characteristic 0 for *many* inputs v_1, v_2.