Improving Secure Server Performance by Re-balancing SSL/TLS Handshakes

Claude Castelluccia, Einar Mykletun, Gene Tsudik
Computer Science Department
School of Information and Computer Science
University of California, Irvine
{ccastell,mykletun,gts}@ics.uci.edu

Abstract

Much of today's distributed computing takes place in a client/server model. Despite advances in fault tolerance – in particular, replication and load distribution – server overload remains to be a major problem. In the Web context, one of the main overload factors is the direct consequence of expensive Public Key operations performed by servers as part of each SSL handshake. Since most SSL-enabled servers use RSA, the burden of performing many costly decryption operations can be very detrimental to server performance. This paper examines a promising technique for re-balancing RSA-based client/server handshakes. This technique facilitates more favorable load distribution by requiring clients to perform more work (as part of encryption) and servers to perform commensurately less work, thus resulting in better SSL throughput. Proposed techniques are based on careful adaptation of variants of **Server-Aided RSA** *originally constructed by Matsumoto, et al. [1]. Experimental results demonstrate that suggested methods (termed* **Client-Aided RSA***) can speed up processing by a factor of between 11 to 19, depending on the RSA key size. This represents a considerable improvement. Furthermore, proposed techniques can be a useful companion tool for SSL Client Puzzles in defense against DoS and DDoS attacks.*

Keywords: Load-balancing, Server-Aided RSA, Denial-of-Service, Server-Aided Secure Computation, Client Puzzles, Hardware Accelerators

1 Introduction

Much of today's distributed computing takes place in a client/server setting. Server overload, whether due to an on-slaught of legitimate client requests or a Denial-of-Service (DoS) attack, is common occurrence in modern client/server environments, such as the Web. Typically, a server becomes swamped under a flood of simultaneous or closely spaced requests, each requiring it to perform some costly computation, e.g., decrypt a key purportedly encrypted by a client.

Techniques for graceful service degradation have been studied in the past and implemented in real-world servers. Traffic management and congestion control literature offers numerous methods for mitigating traffic spikes in routers. Also, advances in fault tolerance – in particular, replication and load distribution – have been very beneficial to Web servers. However, DoS attacks are not thwarted by such measures since their central goal is to deny service to legitimate clients. Moreover, server overload can occur for reasons other than hostile attacks, e.g., a large number of concurrent benign client requests can still overwhelm a popular server. This can result in server crashes or in denial-of-service to clients who are literally left hanging or presented with a familiar "server busy" message.

Scope of Paper: In this paper, we explore one possible approach to alleviating server load in the Web setting. Specifically, we target SSL/TLS client-server handshakes and focus on altering the computational balance (and burden) between SSL clients and servers. This paper makes a contribution by investigating so-called Server-Aided RSA (SA-RSA) techniques as a way of reducing server overload[1]. SA-RSA was originally proposed as a way to reduce load on small devices (primarily smartcards) by farming out some heavy-weight cryptographic computation to more powerful servers – host computers equipped with smartcard readers. We adapt SA-RSA to the SSL/TLS setting by

[1]Contrary to "popular belief", our proposed solution is not subject to the *meet-in-the-middle* attack proposed in [6]

re-assigning the roles: SSL clients become "servers" in SA-RSA parlance and overloaded SSL servers become "weak clients". The resultant *Client-Aided* RSA (CA-RSA) turns out to be surprisingly effective, achieving server speed-ups of between 11 and 19 times over plain SSL, depending on RSA modulus size.

From the outset, we note that there are alternative techniques for speeding up, or reducing load on, SSL/TLS servers, such as employing Elliptic Curve-based cryptosystems. However, we believe that, in the near future, the well-known and time-tested RSA cryptosystem will continue to dominate in SSL/TLS handshake protocol. Therefore, this paper focuses on improving SSL/TLS performance assuming the use of RSA. (Other relevant approaches and techniques, e.g., cryptographic hardware accelerators, are discussed in Section 5.)

Organization of Paper: The rest of the paper is organized as follows. Section 2 overviews SSL/TLS and motivates our work. Section 3 describes our SSL extension, Client-Aided RSA, for speeding up performance of secure servers and presents performance results. Section 4 extends our protocol to protect against DoS attacks. Related work is reviewed in section 5 and section 6 concludes the paper.

2 Overview of SSL/TLS

This section describes the SSL/TLS handshake protocol [2]. In the remainder of the paper, the term "SSL" is used to refer to both SSL and TLS standards. SSL is the most widely used protocol to ensure secure communication over the Internet. It is typically employed by web servers to protect electronic transactions. SSL uses the RSA cryptosystem during an initial client/server handshake to establish a shared symmetric key for use during an SSL session[2].

2.1 SSL Handshake Protocol Description

The simplest version of the SSL handshake (key-establishment) protocol is shown in figure 1 and consists of two communication rounds that contain the following messages and computations:

1. Client sends a "*client hello*" message to server. This indicates that client wants to initialize a SSL/TLS session and the message includes the cipher suites client supports and a random nonce r_c.

[2]Diffie-Hellman is also supported, at least, according to SSL specifications. However, neither Microsoft nor Netscape offer browser support for non-RSA certificates [3].

2. Server responds with a "*server hello*" message that includes server's public-key certificate and a random nonce r_s. It also specifies server's choice of cipher suite from among client's candidates.

3. Client chooses a secret random 48-byte[3] *pre-master secret* x and computes the shared *master secret* k by inputting values x, r_c, r_s into hash function f. It then encrypts x with the server's RSA public key and attaches the ciphertext to a "*client key exchange*" message that is sent to server.

4. Server decrypts the *pre-master secret* using its private RSA key, and uses it to compute the shared *master secret* as $f(x, r_c, r_s)$. To conclude the handshake, server sends a "*server finished*" message that includes a keyed hash of all handshake messages.

The most computationally expensive step in the SSL handshake protocol is the server's RSA private-key decryption. Critical web servers often employ expensive cryptographic hardware to speed-up the decryption process, enabling them to handle more simultaneous SSL handshake requests, and, thereby, more SSL connections. Hardware accelerators and other techniques for speeding up RSA decryptions are discussed in section 5.

2.2 Computational Imbalance

As noted above, the goal of the SSL/TLS handshake is the establishment of a shared client-server key. The most important component of this process is the client's encryption of a (randomly selected) key under the server's RSA public key. The ciphertext is then transmitted to the server which decrypts it and extracts the key. The core of the over-exertion problem is the RSA decryption operation.

RSA is a mature, well-studied and nearly ubiquitous public key encryption method [4]. However, many (perhaps even most) implementations of RSA encryption are computationally lopsided: they use small public exponents, such as: 3, 17, and $2^{16} + 1$. As a result, RSA encryption is relatively cheap, requiring only a few modular multiplications, whereas, corresponding decryption is expensive as it requires a full-blown exponentiation with the private exponent (d). We note that decryption remains expensive even if the well-known CRT (Chinese Remainder Theorem) technique is used to speed it up. This imbalanced arrangement is clearly

[3]Actually, only 46 of the 48 bytes are random. The other 2 bytes contain the SSL version number.

Client		**SSL Server**
r_c, cipher-specs	*client hello* $\longrightarrow$	
	$\longleftarrow$ *server hello*	r_s, cipherspecs
$x \in_R \{0,1\}^{48}$ $k = f(x, r_c, r_s)$ $y = x^e \pmod n$	*client key exchange* $\longrightarrow$	
		$x = y^d \pmod n$ $k = f(x, r_c, r_s)$
	$\longleftarrow$ *server finish*	

Figure 1. The SSL handshake

beneficial for computationally challenged clients, however, it is detrimental to server's connection throughput and general availability.

One possible solution to correct the imbalance is to select the private exponent d to be small, thereby speeding up decryption. However, choosing too small of a value leads to RSA becoming insecure, as shown in [5]. Therefore the performance improvement provided by this solution is limited.

A more drastic approach is to alter the underlying key establishment protocol to have the server generate and encrypt the session key, thus shifting the decryption burden to the client. Besides being a radical change, this would necessitate a client first supplying an RSA public key to the server. If the client's public key is uncertified, the server would need to perform a public key encryption without verifying the supplied public key. This, once again, presents an opportunity for DoS attacks. To require all clients' public keys to be certified is a major burden for clients. Moreover, the server would need to verify a certificate chain for each connection which is an expensive proposition.

However, the main problem with the above approach is that the server would still need to be authenticated. Recall that the key establishment in SSL/TLS serves a dual purpose: in addition to securely transporting a client-selected key to the server, the protocol implicitly authenticates the server. The latter would be lost if the server encrypts the session key for the client; unless, of course, the server signs something which brings us right back to the computational imbalance issue.

3 Client-Aided RSA

The purpose of the above discussion is to motivate techniques for re-balancing the lopsidedness of RSA decryption and, as a result, speeding up decryption operations on the server side. To do so, we focus on the well-known general technique of Server-Aided Secret Computation (SASC) and Server-Aided RSA (SA-RSA), in particular. The original idea of Server-Aided RSA is due to Matsumoto, et al. [1]. Its prime motivation is to off-load expensive RSA signature computation from a weak device (such as a smartcard) to a powerful-but-untrusted server, without exposing any information about the device's private exponent.

In this paper, we flip SA-RSA around to obtain **Client-Aided RSA (CA-RSA)**. The main idea is to shift some computational burden from the server to the clients. Specifically, we want the clients to perform the bulk of the work in RSA decryption, thereby allowing the server to accept and process more incoming requests.

3.1 Protocol Description

We now describe the CA-RSA algorithm. We first introduce the basic version and then extend it to obtain CA-RSA.

3.1.1 Basic Version

We begin by representing the server's private exponent as $d = f_1d_1 + f_2d_2 + ... + f_kd_k \pmod{\phi(n)}$, where the f_i's and d_i's are random vector elements of c and $|n|$ bits, respectively.

The following process take place when a server wants to offload the computation $x^d \pmod n$ to a client:

1. Server sends vector $D = (d_1, d_2, ..., d_k)$ to client.

2. Client computes vector $Z = (z_1, z_2, ..., z_k)$, where $z_i = x^{d_i} \pmod{n}$, and sends it back to server.

3. Finally, server computes $\prod_{i=1}^{k} z_i^{f_i} = \prod_{i=1}^{k} x^{f_i d_i} = x^d \pmod{n}$

The choice of parameters: k, c, and the f_i's is discussed in section 3.4. Note that, assuming that it is computationally difficult to "break" RSA, parameter selection should not introduce any attacks that compromise the security of the above computation by the server, namely $x^d \pmod{n}$. An attacker can attempt to exhaust all possible vector values f_i thereby deriving d. Thus, a minimal requirement for c and k is that a brute force attack (which requires $2^{c \times k}$ steps) should be as difficult as breaking underlying RSA[4].

3.1.2 CA-RSA

CA-RSA improves upon the performance of the basic scheme by taking advantage of the Chinese Remainder Theorem (CRT). Quisquater and Couvreur [7] demonstrated how RSA secret key exponentiations could be sped up with CRT. The technique works as follows: Let $d_p = d \mod (p-1)$ and $d_q = d \mod (q-1)$. For $M_p = M^{d_p} \pmod{p}$ and $M_q = M^{d_q} \pmod{q}$, we have $M^d = M_p \times n_p + M_q \times n_q \pmod{n}$, where $n_p = q \times (q^{-1} \pmod{p})$ and $n_q = p \times (p^{-1} \pmod{q})$. Because n_p and n_q can be pre-computed, and, since exponentiations mod p or q are more efficient to compute than those mod n, we can expect an approximate factor of 4 speed-up of private-key operations [8] when using the CRT.

In CA-RSA, the server initially pre-computes d_p, d_q, n_p and n_q, where n_p and n_q are derived as described above:

$$d_p = \sum_{i=1}^{k} f_i d_i \mod (p-1), \quad d_q = \sum_{i=1}^{k} g_i d_i \mod (q-1)$$

All f_i's and g_i's are random c-bit values.

The following takes place when a server wants to offload the computation of $x^d \pmod{n}$ to a client:

1. Server sends vector $D = (d_1, d_2, ..., d_k)$ to the Client.

2. Client computes vector $Z = (z_1, z_2, ..., z_k)$, where $z_i = x^{d_i} \pmod{n}$, and sends it back to server.

3. Server computes intermediary values $M_p = \prod_{i=1}^{k} z_i^{f_i} \pmod{p}$ and $M_q = \prod_{i=1}^{k} z_i^{g_i} \pmod{q}$. Finally, $x^d = M_p n_p + M_q n_q \pmod{n}$.

[4] Actually, only $2^{c \times k/2}$ steps are needed to break this basic scheme via the classical *meet-in-the-middle* attack [6]. However, the attack in [6] **does not** apply to the CA-RSA protocol described in the following section.

3.2 Incorporating CA-RSA into the SSL Handshake

We now describe the modifications to the SSL Handshake protocol necessary to incorporate CA-RSA. The *client hello* and *server hello* messages remain unchanged, although the server's certificate (which is sent as part of the *server hello* message) now includes the vector $D = (d_1, d_2, ..., d_k)$. The client chooses a random value x, which is then used to derive the SSL session key, and uses the server's public exponent to encrypt it: $y = x^e \pmod{n}$. Next, the client uses D to construct a vector Z by computing the individual vector elements $z_i = y^{d_i} \pmod{n}$, for $1 \le i \le k$. This vector is included in the *client key exchange* message. The server, upon receiving this message, performs its CRT computations and derives $y^d = (x^e)^d = x \pmod{n}$. The remainder of the handshake remains unchanged. Figure 2 shows the modified protocol.

3.3 Security and Parameter Selection

This section discusses security considerations and parameter selection issues for CA-RSA.

Two Server-Aided RSA schemes were originally proposed by Matsumoto et al. [1]: RSA-S1 and RSA-S2. They correspond to "Simple CA-RSA" and CA-RSA algorithms, respectively. In fact, CA-RSA is almost identical to RSA-S2, except the roles of the client and server are reversed.

Initially, RSA-S1 and RSA-S2 used binary exponent values for f_i and g_i. These versions of Server-Aided RSA were soon subject to attacks. Recall that, in RSA-S1, the private exponent d is represented as $d = \sum_{i=1}^{k} f_i d_i \pmod{\phi(n)}$, where the f_i's are randomly selected c-bit elements ($c = 1$ when binary exponents are used). Once vector $D = (d_1, d_2, ..., d_k)$ is sent to the client (as part of the protocol), the secrecy of private exponent d relies upon the secrecy of the f_i's. Using binary values for the f_i's allows for simple, but effective, attacks [9, 10].

Subsequent incarnations (RSA-S1M and RSA-S2M) by Matsumoto, et al. [11] were also attacked by Lim and Lee [12]. This led to the development of parameter selection guidelines for RSA-S1 and RSA-S2 [13, 14]. The goals of these guidelines were to protect against known vulnerabilities and to suggest parameter values that would withstand brute force attacks aimed at

Client		**SSL Server**
r_c, cipher-specs	$\xrightarrow{\textit{client hello}}$	
	$\xleftarrow{\textit{server hello}}$	r_s, cipherspecs
$x \in_R \{0,1\}^{48}$ $k = f(x, r_c, r_s)$ $y = x^e \pmod n$ $z_i = y^{d_i} \pmod n$ $Z = (z_1, ..., z_k)$		
	$\xrightarrow{\textit{client key exchange}}$	
		$M_p = \prod_{i=1}^{k} z_i^{f_i} \pmod p$ $M_q = \prod_{i=1}^{k} z_i^{g_i} \pmod q$ $x = y^d = M_p n_p + M_q n_q \pmod n$ $k = f(x, r_c, r_s)$
	$\xleftarrow{\textit{server finish}}$	

Figure 2. Incorporating CA-RSA into the SSL handshake protocol

finding the server's private RSA exponent d. In summary, when used together with the suggested parameter guidelines, CA-RSA (i.e., RSA-S2) has not been successfully attacked.[5]

For CA-RSA, the guidelines required for it to be infeasible to deduce values d_p or d_q via brute force, since an attacker with knowledge of either one would be able to factor modulus n and thereby break RSA. Given a vector $D = (d_1, d_2, ..., d_k)$, a search through all possible values of F (respectively G) would reveal d_p (respectively d_q). Because there are k c-bit vector elements, the guidelines mandated that the search space of $2^{c \times k}$ values be large enough to prevent such an exhaustive search.

When choosing CA-RSA parameters, we specifically selected the values c and k such as to meet the requirements set forth in the guidelines while making the difficulty of exhausting the resulting search space at least equivalent to (or harder than) breaking the underlying RSA cryptosystem. As is well-known, the strength of the RSA cryptosystem, when correctly instantiated, depends upon the key (modulus) size. Currently, 1024-bit keys are common, however, based upon projected advances in computing power, experts in the cryptography research community recommend using larger values for longer-term security.

Exhaustive search of $2^{c \times k}$ values is equivalent to searching for all possible keys in a symmetric-key cryptosystem (for example DES, AES or Blowfish). Thus, based upon the RSA key size used, we need to determine symmetric key size that would provide equivalent security. Lenstra and Verheul give formulas for determining such keys in their well-known work on cryptographic key size selection [19]. They use historical cryptanalysis developments and projected computing powers to develop hypotheses and create formulas for choosing cryptographic key sizes, depending upon how far into the future the cryptosystems are to remain secure. Since their formulas cover both symmetric and asymmetric cryptosystems, the results are applicable for our purposes. Based on their formulas, RSA with 1024- and 1536-bit keys would be roughly equal in strength to a symmetric-key cryptosystem with 72- and 80-bit keys, respectively.

3.4 Performance

This subsection describes our experimental results

3.4.1 Experiment set-up

We measured the speedup in the execution time of RSA decryptions when using CA-RSA instead of plain RSA (with CRT). As noted in section 2, the most computationally expensive operation in the SSL handshake protocol is the server's private key decryption. Therefore, we determine an upper bound on the number of SSL requests by measuring the number of RSA decryp-

[5] Certain other variations of Server-Aided RSA [15, 16] were later found suspectible to lattice reduction attacks [17, 18].

tions a server can perform within a given time frame. Our hardware platform was a 1.7 Ghz Intel Celeron with 256 MB RAM running Red Hat 9.0 Linux. The scripts were written using the OpenSSL cryptographic library (version 0.9.7). RSA keys of 1024, 1536 and 2048 bits were used so as to test CA-RSA performance with both current and future security parameters.

3.4.2 Results

Table 1 lists the average decryption time (in msecs) for the three moduli with both plain RSA and CA-RSA.

Table 1. Average decryption time (msec):

Key size	RSA	CA-RSA	$c \times k$	Improvement
1024	7.05	0.62	72	11.33
1536	19.79	1.25	80	15.76
2048	44.22	2.31	88	19.12

These results show CA-RSA speedups of 11.3, 15.8 and 19.1 times (as opposed to plain RSA) for 1024-, 1536- and 2048-bit keys, respectively. Expected theoretical speed-ups are 13, 17.8 and 21.7, respectively. These results compare favorably with another technique aimed at speeding up RSA decryptions – SSL batching proposed by Shacham and Boneh [20] – which achieves a factor of 2.5 speed-up for 1024-bit RSA keys.

As described in section 3.3, the CA-RSA values c and k were selected based upon key size formulas in [19], such that $c \times k$ corresponds to a symmetric key comparable in strength to the corresponding RSA key. Specifically, for 1024-, 1536- and 2048-bit keys, $c \times k$ was set to 72, 80 and 88 bits, respectively. The results mean that a server with a 1024-bit RSA key can perform approximately 11 times as many decryption operations per second. A secure web server achieving such speedups (of one order in magnitude) becomes comparable to hardware-accelerated SSL servers when using CA-RSA as opposed to plain RSA.

For CA-RSA, the optimal parameter selection strategy is to minimize k and thereby maximize c. The parameter selection guidelines (section 3.3) specify the smallest possible value of k to be 2, and c is therefore set to 36, 40 and 44 bits for the 72, 80 and 88 bit keys, respectively. Figure 3 shows how decryption time varies depending on the distribution of bits between parameters c and k, while maintaining the property that $c \times k = 72, 80$ and 88 for the RSA keys of equivalent strength.

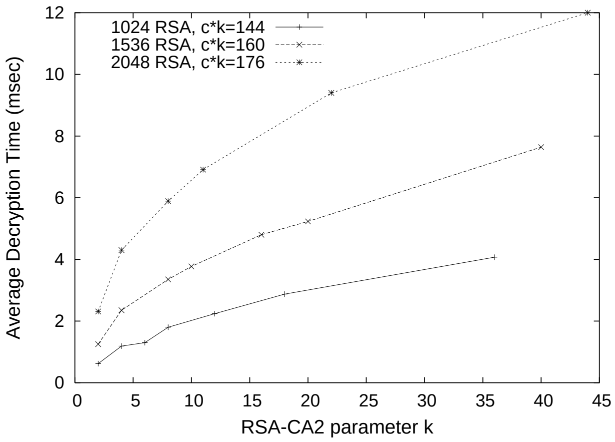

Figure 3. CA-RSA decryption time (msec) when varying parameters c and k

3.5 Discussion

Althought CA-RSA reduces the computation load at the server, it introduces certain computation and bandwidth costs at the clients.

- **Computation:** A client with a computing environment described in section 3.4.1 would incur added computational costs of approximately 21.9, 66 and 150.9 msecs when computing the elements of vector Z for 1024-, 1536- and 2048-bit RSA keys, respectively. We believe that this added computation cost is negligible and acceptable for most clients. For weak computational devices offloading techniques such as the one described in [21] could be used.

- **Bandwidth:** The bandwidth overhead associated with the *client key exchange* message now includes the vector Z. It contains k $|n|$-bit elements, where n is the RSA modulus. Recall that, with plain RSA, the client only sends $y = x^e \pmod{n}$. Therefore, with $k = 2$ (chosen for optimal performance), the resulting extra bandwidth translates into $|n|$ bits. This corresponds to less than one ethernet frame (1500 bytes)[6]. Furthermore, adding such a small number of bits in the SSL handshake does not have a significant impact on performance. In fact, as shown in [22], SSL is purely CPU bounded and optimizations intended to reduce network bandwidth have little effect on server throughput.

[6]On a related note, the vector D needs to be added to the server's public-key certificate. This can be achieved by including it as an extension field in X.509v3 format.

Note that one reasonable strategy is to use our re-balancing technique only when the server gets overloaded, and use regular SSL otherwise. In this case, the previously described extra computation and bandwidth costs only occurs occasionally.

4 SSL Speedup and DoS Protection

Server overload is either due to an on-slaught of legitimate client requests or a Denial-of-Service (DoS) attack. A server can become swamped under a flood of simultaneous of closely spaced requests, each requiring it to perform some costly computation, e.g., decrypt a key purportedly encrypted by a client. This makes SSL servers prime targets of DoS and DDoS attacks.

The protocol described in section 3 increases the number of SSL connections that the server can handle by re-balancing the computations between the server and client. This makes the job of a potential DoS attacker more difficult, but a resourceful attacker can still achieve his goal by increasing his resources accordingly. In addition to CA-RSA, which helps the server by reducing its computational load, we need a mechanism that that also makes the job of the attackers more difficult.

We view the DoS menace as being two-fold: (1) the adversary overwhelms the server with a sheer number of gratuitous service requests and, (2) the adversary over-exerts the server by forcing it to perform many heavy-weight cryptographic operations. Although client puzzles *alleviate* both problems (see section 4.1), they do not completely solve either. Arguably, there might be simply no way to solve the former since a determined and resourceful adversary will always be able to flood the server with a storm of requests (even if they are quickly filtered out). On the other hand, a computationally powerful adversary can efficiently dispense with the minor "inconvenience" posed by puzzles and similar techniques; such an adversary can still force the server to perform many expensive cryptographic operations and thus render the server unavailable to legitimate clients.

An attacker who attempts to incapacitate a secure web server needs only to initiate as many SSL handshake requests per second as the number of RSA decryptions the server can perform per second. (For example, on our test server, one RSA decryption takes approximately 7 msec, thus, it can perform at most 142 decryptions per second. However, higher-end web servers can perform up to $4,400$ RSA decryptions per second [23].) The feasibility of such DoS attacks is partly because a client can request the server to perform many RSA decryptions without performing any significant amount of work itself. A possible remedy is to: (1) ask the client to perform a certain amount of additional work prior to triggering the server to decrypt, and/or (2) speed up the decryption operation on the server side such that a DoS attack requires greater resources. Our solution combines the above two properties: it requires a client to perform additional computation which then lessens the load on the server, thus allowing it to perform more RSA decryptions and accept/process more incoming connections.

4.1 Client Puzzles and SSL

Juels and Brainard introduced the use of client puzzles as a cryptographic countermeasure to protect against DoS attacks [24]. Dean and Stubblefield subsequently proposed using client puzzles to specifically defend web servers running the SSL protocol [25]. Their scheme requires a client to solve a given puzzle before being able to establish an SSL session with a server. This forces the client to perform a certain amount of computational work prior to requesting the server to carry out expensive operations (such as RSA decryptions). That way, a DoS attack becomes more computationally demanding to execute as clients can no longer freely trigger RSA decryptions. The type of client puzzle they use consists of inverting a hash function when given the hash digest and a certain portion of the pre-image.

The addition of client puzzles does not alter the message flow in the SSL handshake protocol, but does require two of the messages to be extended. After the client has sent its *client hello* message, the server chooses a random a-bit value s and inputs it to a cryptograhic hash function. It then includes the hash digest $t = hash(s)$ along with the b first bits of s (where $b < a$) to the client in the *server hello* message. Using these b bits, the client solves the "client puzzle" via brute-force and finds a value s' that hashes to the desired t. With knowledge of the first b pre-image bits, the client only needs to attempt approximately 2^{a-b} candidate values before finding a valid solution s' that satisfies $t = hash(s\text{'})$.

The client then includes s' in its *client key exchange* message. Only if s' verifies - i.e, it is of correct length and its hash output is t - will the server proceed with the SSL handshake and decrypt the encrypted session key submitted by the client.

The computational cost of a hash computation is almost negligible when compared to an RSA decryption (a hash is about 3 to 4 orders of magnitude faster to compute), so the addition of puzzle verification step adds a very minor server side overhead. The amount

Client **SSL Server**

r_c, cipher-specs

client hello $\longrightarrow$

r_s, cipherspecs
$s \in_R \{0,1\}^a$, $t = hash(s)$
r_s, t, $s_{(b)}$, cipherspecs

$\longleftarrow$ *server hello*

s' s.t. $t = h(s')$
$x \in_R \{0,1\}^{48}$
$k = f(x, r_c, r_s)$
$y = x^e \pmod n$
$z_i = y^{d_i} \pmod n$
$Z = (z_1, ..., z_k)$, s'

client key exchange $\longrightarrow$

verify $t = h(s')$
$M_p = \prod_{i=1}^{k} z_i^{f_i} \pmod p$
$M_q = \prod_{i=1}^{k} z_i^{g_i} \pmod q$
$x = y^d = M_p n_p + M_q n_q \pmod n$
$k = f(x, r_c, r_s)$

$\longleftarrow$ *server finish*

Figure 4. Incorporating CA-RSA together with Client Puzzles into the SSL handshake protocol

of work needed to be done by the client in order to solve the puzzle depends upon its computing resources and, more importantly, the number of unknown bits in the pre-image value sent by the server.

The addition of client puzzles to the SSL handshake protocol has the advantage of making DoS attacks more elaborate to carry out. A single client machine is no longer able to easily overload an SSL server by sending consecutive SSL initiation requests, as it would need to solve the appropriate client puzzles, thereby limiting the number of valid requests it could send per second. A more noticeable side effect of utilizing client-puzzles is that client browser software needs to be modified to make it work with the puzzles during the SSL handshake protocol.

4.2 Combining Client Puzzles and CA-RSA

We now sketch out a way of combining client puzzles and CA-RSA. When a client initiates a session with a secure web server, it receives d_i values (included in the server's certificate) and a puzzle as part of the SSL handshake. The client solves the puzzle, computes z_i as required by CA-RSA and returns these values, along with the puzzle solution, to the server. If the server successfully verifies the puzzle solution, it performs the CA-RSA partial decryption needed to compute the session key. Figure 4 gives an overview of the protocol. The notation $s_{(b)}$ refers to the first b bits of the pre-image value s. A client response without a valid puzzle solution is simply ignored.

Of course, a malicious client can solve the puzzle and still send bogus z_i's to the server. However, the amount of wasted effort is much less – 11 times smaller for 1024-bit RSA keys – than in case when only client puzzles are used (as in [25]).

Furthermore, the extra work resulting from CA-RSA by the client effectively adds to the cost of solving the puzzle, but does not affect an adversary since he can skip the CA-RSA step and only work on the puzzle. However, as shown in figure 5, the CA-RSA cost quickly becomes negligible compared to client puzzle cost as the puzzle size increases. More precisely, when the attack is not severe, and therefore the puzzle size is small, the added cost to a legitimate client is very small. But when the intensity of the attack increases, and subsequently the puzzle size increases, the extra cost of CA-RSA fades away.

Both the client puzzle and our CA-RSA mechanisms aim at solving the problem of server overload in different but complementary ways:

- Puzzles slow DoS attacks by forcing attackers to

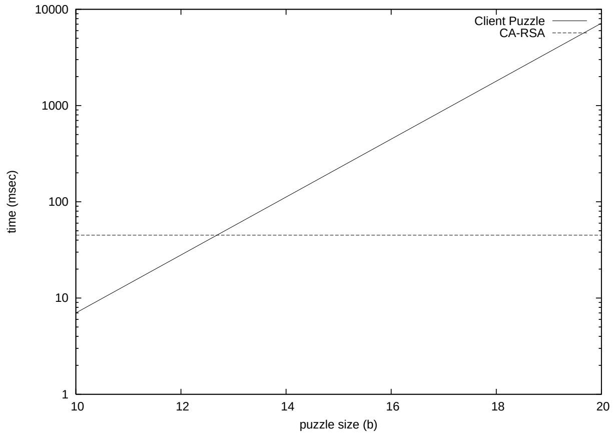

Figure 5. Comparing client computational cost of client puzzles and CA-RSA

perform some work before the server commits resources.

- CA-RSA reduces server load by outsourcing some of the its computation to the clients, allowing the server to accommodate more SSL requests.

In summary, the combination of client puzzles and CA-RSA offers an effective countermeasure to server overload and DoS/DDoS attacks.

5 Related Work

Techniques for speeding up SSL transactions typically aim to accelerate RSA decryptions and can be classified into two categories: dedicated cryptographic hardware accelerators and non-standard RSA decryption techniques.

5.1 Hardware Accelerators

SSL hardware accelerators are dedicated modular arithmetic processing units aimed at speeding up RSA computations. One example of a hardware accelerator is the SonicWALL SSL-RX [23] which is claimed to achieve up to 4,400 RSA decryptions per second, and comes with a price tag of around $14,000.

Accelerators range widely in both speed and price. They also often give a smaller than expected increase in SSL throughput. In [22], Coarfa, et al. analyze the performance of SSL and conclude that hardware accelerators are not as effective as originally thought: : depending on the workload, one might only achieve a factor of 2 speedup. Specifically, when session re-use is high, resulting in few full SSL handshakes, only a modest gain in SSL throughput is actually achieved: approximately a factor of 2 speedup. Authors suggest that, instead of purchasing a relatively expensive cryptographic accelerator, a better choice would be to invest in a faster CPU to better handle encryption of application data during SSL sessions. This conclusion is inline with our work that does not require any specific hardware but would benefit from a more powerful CPU.

Berson, et al. [26] propose offering cryptographic operations, such as modular exponentiations, as a network service. A so-called cryptoserver would be equipped with a multiple hardware accelerators and its services would be shared amongst many clients. Although trust is a major concern in this model, there are some application settings where the cryptoserver might be in the same security perimeter as its users (e.g., web servers). An example would be a cryptoserver supporting SSL for a group of secure web servers that are all part of the same organization. A similar idea is due to Mraz [21] where certain portions of the SSL protocol – RSA processing and bulk encryption – are offloaded to an array of special-purpose (SSL handshake-optimized) servers.

5.2 RSA Speedup Techniques

Another approach to speeding up SSL handshakes involves techniques for accelerating RSA decryptions without the use of specialized hardware. We begin by describing the seminal work by Shacham and Boneh which proposes three methods for faster RSA decryptions [20, 27]. From an encryptor's (i.e., an SSL client's) perspective, all three methods are backward compatible with standard RSA. Also, all speedups discussed below are based on 1024-bit RSA and are relative to the cost of performing plain RSA decryptions.

The first technique is based on multi-factor RSA moduli. Specifically, the RSA setting is that of multi-prime and multi-power moduli, where $n = pqr$ or $n = p^2q$ (instead of the usual $n = pq$), and decryption is performed using CRT and Hensel lifting [28], respectively. One can expect theoretical speedups of around 2.25 with $n = pqr$ and 3.38 for $n = p^2q$. Experiments show real speedups to be around 1.73 and 2.3, respectively.

Similar to CA-RSA, the second method – rebalanced RSA – shifts the workload to the encryptor. It is a variant of an earlier technique by Weiner [29]. Specifically, d is chosen to be close to n such that both $d \mod (p-1)$ and $d \mod (q-1)$ are small integers. The resulting public exponent e also becomes close to n, which is much larger than typical values (i.e., $e = 3, 17$, or 65537). It is in fact so large that Microsoft Internet Explorer (IE) cannot accept it; IE allots a maximum of 32 bits for the public exponent e. Rebalanced RSA offers the theoretical speedup of 3.6 but the actual speedup is 3.2.

The third technique – batch RSA – is based on Fiat's Batch RSA which, in turn, relies on simultaneous exponentiations [30]. This technique offers the speedup factor of 2.5. Batch RSA uses a batching parameter b that defines the number of ciphertexts needed in order to batch-decrypt. (Typically, b is set to 4 for optimal performance.) Each SSL server needs b RSA public key certificates, each with identical modulus but different public and private keys. When it receives b pending SSL handshake requests, each based upon one of the certificates, the server takes advantage of the batching technique and performs b decryptions in less time than if it executed them sequentially. A heavily loaded web server using a round-robin strategy when sending certificates to clients would incur minimal latency before receiving 4 SSL handshake requests with distinct certificates.

We now mention one other technique for speeding up RSA computations. In [31], Lim and Lee discuss using RSA precomputations in order to speed up modular exponentiation. A tradeoff is made between storage space (committed to precomputed values) and computation time, with more pre-computations resulting in more efficient exponentiations. This technique outperforms other modular exponentiation algorithms such as Square-and-Multiply and BGMW methods [8].

5.3 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) has been gaining attention as an attractive alternative to more traditional public-key cryptosystems. ECC offers certain advantages, notably, it can provide an equivalent level of security as other public key methods with smaller key sizes and faster computation. Currently, one of the main reasons hindering wider acceptance of ECC is the existence of multiple patents.

In [32] Gupta, et al. analyze achievable performance gains when using ECC to speed up SSL. Their results show that the performance gain of ECC over RSA increases for larger key sizes. In comparison with currently common 1024-bit RSA setting, they measure a speedup factor of the server RSA decryption time of 2.4 when using 160-bit ECC keys which offers equivalent level of security. Since our scheme provides a performance gain of about 11 compared to the regular RSA setting, it would outperform ECC's performance by a ratio of 4. In order words, ECC benefits, such as short keys, do not help to solve our problem of server overload.

6 Conclusion

We proposed a variation of Server-Aided RSA for rebalancing RSA-based client/server handshakes, specifically targeting SSL/TLS. Clients are required to perform "useful" work, thereby freeing up the server's resources and allowing it to perform commensurately less work, thus resulting in better throughput. We stress that our approach is not an alternative, but a supplement, to client puzzles in defense against DoS and DDoS attacks. Experimental results demonstrate that our Client-Aided RSA solution achieves substantial performance improvements over the basic RSA (with CRT) decryption algorithm, namely speedups of between 11 and 19, depending on the RSA key size. A secure web server achieving software-speedups of one order in magnitude becomes competitive with hardware-accelerated SSL servers.

References

[1] T. Matsumoto, K. Kato, and H. Imai, "Speeding up Secret Computations with Insecure Auxiliary Devices," *Proceedings of Crypto '88*, pp. 497–506, 1988.

[2] Private communication, "Private communication with David Wagner," 2005.

[3] Network Working Group, "RFC 2246 - The TLS Protocol Version 1.0," *Internet RFC/STD/FYI/BCP Archives*, 1999, `http://www.faqs.org/rfcs/rfc2246.html`.

[4] BEA WebLogic, "BEA WebLogic Server Frequently Asked Questions," `http://e-docs.bea.com/wls/docs60/faq/security.html`.

[5] Ron L. Rivest, Adi Shamir, and Leonard M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[6] D. Boneh and G. Durfee, "Cryptanalysis of RSA with Private Key d Less than $n^{0.292}$," *IEEE Transactions on Information Theory*, vol. 46, pp. 1339–1349, 2000.

[7] J. Quisquater and C. Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystem," *Electronic Letters*, vol. 18, pp. 1905–907, 1982.

[8] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone, "Handbook of Applied Cryptography," *The CRC Press series on discrete mathematics and its applications*, 1997.

[9] B. Pfitzmann and M. Waidner, "Attacks on Protocols for Server-Aided RSA Computations," *Proceedings of Eurocrypt '92*, pp. 153–162, 1992.

[10] R. J. Anderson, "Attack on Server Assisted Authentication Protocols," *Electronic Letters*, vol. 28, pp. 1473, 1992.

[11] T. Matsumoto, H. Imai, C. S. Laih, and S. M. Yen, "On verifiable implicit asking protocols for RSA computation," *Advances in Cryptology - Proceedings of Auscrypt '92*, vol. 718, pp. 296–307, 1992.

[12] C. H. Lim and P. J. Lee, "Security and performance of server-aided RSA computation protocols," *Advances in Cryptology - CRYPTO '95*, pp. 70–83, 1995.

[13] J. Burns and C. Mitchell, "Parameter Selection for Server-Aided RSA Computation Schemes," *IEEE Transactions on Computing*, vol. 43, pp. 163–174, 1994.

[14] C. Laih and F. Tu, "Remarks on Parameter Selection for Server-Aided Secret RSA Computation Schemes," *International Workshops on Parallel Processing*, pp. 167–173, 1999.

[15] P. Béguin and J.J. Quisquater, "Fast Server-Aided RSA Signatures Secure Against Active Attacks," *Advances in Cryptology - CRYPTO '95*, pp. 57–69, 1995.

[16] S. Hong, J. Shin, and H. Lee-Kwang, "A new approach to server-aided secret computation," *International Conference on Information Security and Cryptology*, pp. 33–45, 1998.

[17] J. Merkle, "Multi-round passive attacks on server-aided RSA protocols," *Proceedings of the 7th ACM conference on Computer end Communications security*, pp. 102–107, 2000.

[18] P. Nguyen and I. Shparlinski, "On the insecurity of a server-aided RSA protocol," *Proceedings of Asiacrypt '01*, vol. 2248, pp. 21–35, 2001.

[19] Arjen K. Lenstra and Eric R. Verheul, "Selecting cryptographic key sizes," *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, vol. 14, no. 4, pp. 255–293, 2001.

[20] H. Shacham and D. Boneh, "Improving SSL Handshake Performance via Batching," *Proceedings of RSA 2001*, vol. 2020, pp. 28–43, 2001.

[21] Ron Mraz, "Secure Blue: An Architecture for a High Volume SSL Internet Server," *17th Annual Computer Security Applications Conference*, 2001.

[22] Cristian Coarfa, Peter Druschel and Dan S. Wallach, "Performance Analysis of TLS Web Servers," *9th Network and Systems Security Symposium*, pp. 553–558, 2002.

[23] SonicWALL, "SonicWALLE SSL-RX," `http://www.sonicwall.com/products/sslrx.html`.

[24] Ari Juels and John Brainard, "Client Puzzles: A Cryptographic Defense Against Connection Depletion," *5th Network and Systems Security Symposium*, pp. 151–165, 1999.

[25] D. Dean and A. Stubblefield, "Using Client Puzzles to Protect TLS," *Proceedings of the USENIX Security Symposium*, 2001.

[26] T. Berson, D. Dean, M. Franklin, D. Smetters, and M. Spreitzer, "Cryptography as a Network Service," *7th Network and Systems Security Symposium*, 2001.

[27] D. Boneh and H. Shacham, "Fast Variants of RSA," *CryptoBytes (RSA Laboratories)*, vol. 5, pp. 1–9, 2002.

[28] H. Cohen, "A Course in Computational Algebraic Number Theory," *Graduate Texts in Mathematics*, vol. 138, pp. 6, 1996.

[29] M. Weiner, "Cryptanalysis of Short RSA Secret Exponents," *IEEE Transactions on Information Theory*, vol. 36(3), pp. 553–558, 1990.

[30] A. Fiat, "Batch RSA," *Proceedings of Crypto '89*, pp. 175–185, 1989.

[31] C. H. Lim and P. J. Lee, "More Flexible Exponentiation with Precomputation," *Advances in Cryptology - CRYPTO '94*, pp. 95–107, 1994.

[32] Vipul Gupta and Douglas Stebila and Stephen Fung, "Speeding up Secure Web Transactions Using Elliptic Curve Cryptgraphy," *11th Network and Systems Security Symposium*, pp. 231–239, 2004.