Anatomy and Performance of SSL Processing

Li Zhao[†], Ravi Iyer[‡], Srihari Makineni[‡], Laxmi Bhuyan[†]
[†]*Computer Science Department*
University of California, Riverside
{zhao, bhuyan}@cs.ucr.edu
[‡]*Communications Technology Lab*
Intel Corporation
{ravishankar.iyer, srihari.makineni}@intel.com

Abstract

A wide spectrum of e-commerce (B2B/B2C), banking, financial trading and other business applications require the exchange of data to be highly secure. The Secure Sockets Layer (SSL) protocol provides the essential ingredients of secure communications – privacy, integrity and authentication. Though it is well-understood that security always comes at the cost of performance, these costs depend on the cryptographic algorithms. In this paper, we present a detailed description of the anatomy of a secure session. We analyze the time spent on the various cryptographic operations (symmetric, asymmetric and hashing) during the session negotiation and data transfer. We then analyze the most frequently used cryptographic algorithms (RSA, AES, DES, 3DES, RC4, MD5 and SHA-1). We determine the key components of these algorithms (setting up key schedules, encryption rounds, substitutions, permutations, etc) and determine where most of the time is spent. We also provide an architectural analysis of these algorithms, show the frequently executed instructions and discuss the ISA / hardware support that may be beneficial to improving SSL performance. We believe that the performance data presented in this paper will be useful to performance analysts and processor architects to help accelerate SSL performance in future processors.

1. Introduction

The World Wide Web, one of the most popular Internet applications, provides an infrastructure for exchanging data in a client-server environment. This has boosted the rapidly expanding e-commerce and on-line banking systems. One of the most important issues in these systems is security. To address this problem, the Secure Sockets Layer (SSL) [7], Transport Layer Security (TLS) [5] protocol and the Internet Protocol Security Protocol (IPSEC) [11] have been developed and employed to provide secure communications between two applications that run on public domains such as Internet. Although SSL/TLS protocol and IPSEC are situated in different layers (session and network layer respectively), they have common components for security issues. In this paper, we focus on the SSL protocol and do an in-depth analysis of the performance overhead and the execution characteristics.

As secure communications become more important, researchers have been studying the overhead of secure processing and proposing various architectural optimizations for acceleration. In [10], K.Kant et al. studied the performance and architectural impact of SSL on web servers. They showed the overall performance behavior of web server applications when using HTTP versus HTTPS. While this showed the overhead of SSL, it did not present a detailed breakdown of the secure session to show where the maximum performance overhead came from. Other studies [2][12][20] have focused on cryptographic algorithms and proposed optimizations for accelerating these crypto operations. Recently, crypto units [8] have been added to the IXP2850 network processors. The scope of our paper is to profile SSL processing, show how the crypto algorithms affect SSL performance and point out architectural improvements that can be made.

We start by analyzing the overall effect of SSL performance in a web server environment. We show that SSL processing consumes about 70% of an HTTPS transaction. We break the time spent into crypto and non-crypto portions and show that the non-crypto SSL processing takes a negligible fraction of the time. The significant overhead by SSL is mainly due to its crypto operations, which include asymmetric cryptography, symmetric cryptography and hash functions. We then present a detailed anatomy of SSL processing by analyzing and measuring the two major phases – session negotiation (or handshake) and bulk data transfer. Having exposed the anatomy, we then focus our time on examining the frequently used crypto operations. We study their architectural characteristics like CPI, path length and frequently used instructions. For each cryptographic algorithm, we present which underlying operations (substitutions, permutations, encryption rounds, key schedule initialization, etc) take a most amount of time. Based on our observations, we present and discuss hardware support that may be beneficial to improving crypto performance and SSL performance in the future.

The rest of the paper is organized as follows. The background is presented in section 2. The methodology

for our experiments is described in section 3. Section 4 shows the measurement data for SSL processing and crypto operations. We also present the anatomy of SSL handshake and study each component in detail. In section 5, the main crypto operations that contribute to the SSL processing are analyzed. Section 6 studies the architectural characteristics of the crypto operations and discusses hardware support can be used to accelerate SSL processing. The conclusions and future work are described in section 7.

2. Background

Secure communication between two systems (a client and a server for instance) can be achieved if the following three aspects are guaranteed: (1) Privacy -- to ensure that the data exchanged cannot be viewed by a third party, (2) Integrity -- to ensure that the data are not modified along the way transferred and (3) Authentication -- to ensure that the end systems are indeed the systems that they say they are. The SSL protocol, which sits in the session layer, accomplishes this in two phases: (1) a session negotiation phase and (2) bulk data transfer phase in encrypted form.

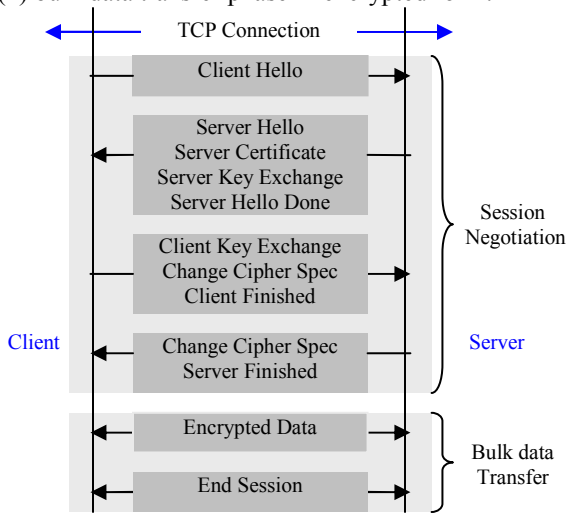

Figure 1. Description of the SSL protocol flow

Figure 1 presents the messages exchanged in these two phases. In the session negotiation (handshake) phase, the client starts the session by sending a "client hello" message. All the subsequent messages exchanged between the client and the server are used to negotiate the cipher suite (what crypto algorithm will be used), an session id (used to resume a previous session), certificates (usually only the server is authenticated), and secret keys for the bulk data transfer. The "finish" messages are sent to finish the session negotiation phase.

In the bulk data transfer phase, the data exchanged between the client and the server is encrypted/decrypted based on the chosen crypto algorithm with the established private keys. Each message is also appended with a message authentication code (MAC) to ensure its integrity. The MAC calculation is based on the secure hash function, a hash function performed on the transferred data together with private keys.

The three main crypto operations used in SSL protocol are: asymmetric encryption (public key encryption), symmetric encryption (private key encryption) and hashing. Asymmetric encryption algorithms like RSA [13] and Diffie-Hellman [6] are used in the handshake phase to exchange secret keys between the server and the client. Symmetric encryption algorithms like AES [14] and DES [13] are used to encrypt/decrypt the data exchanged in the bulk data transfer phase. One-way hash functions like MD5 [13] and SHA-1 [15] are used to guarantee the integrity of the exchanged data. It is used in both the handshake phase as well as the bulk data transfer phase.

For our evaluation in this paper, we choose the following widely used crypto algorithms from each of the above three categories. Below, we briefly introduce these algorithms and discuss their basic concepts.

(1) Public key encryption: RSA is one of the most commonly used public key encryption algorithms. The basic idea is as follows. First two large prime numbers p and q are chosen. Then e is chosen on the condition that 1) e is less than pq and 2) e and (p-1)(q-1) have no prime factors in common. Next d is computed in such a way that (de-1) is evenly divisible by (p-1)(q-1). Now the public key consists the pair (N, e) where N=pq, and the private key consists the triplet (p,q,d). The encryption and decryption functions are the following:

$$C = T^e \bmod N \; ; \; T = C^d \bmod N$$

(C = cipher text, T = plain text)

(2) Private key encryption: Unlike public key encryption, this class of encryption algorithms uses the same key for both the encryption and decryption. There are two types of private key encryption. One is stream cipher encryption (e.g. RC4 [13]), where the plaintext is encrypted in one bit or byte at a time. The input stream is XORed with a random sequence which is generated by a cryptographically-secure keyed pseudorandom number generator. The other is block cipher encryption (e.g. AES, DES/3DES), where encryption is performed in larger units or blocks of data. Data in the block is encrypted using methods like diffusion, substitution and transposition. For the block cipher encryption, one of the most popular modes is chaining-block-cipher (CBC) mode. In this mode, the plain text is XORed with the previous cipher block before it is encrypted. This ensures a dependency between blocks of data within the message and removes the potential for parallelism across individual blocks of data.

(3) Hash Functions: MD5 and SHA-1 are the

widely used hash functions in SSL. These algorithms take a variable length input data and generate a message digest (128 bits for MD5 and 160 bits for SHA-1).

In this paper, we will present a detailed analysis of the time spent in each of these cryptographic operations for SSL sessions. In addition, we will also present a detailed analysis of where the majority of the time is spent within these cryptographic operations.

3. Methodology

In this section, we discuss the methodology we use for evaluating the SSL processing and the crypto operations. The purpose of our experiments is the following: 1) Isolate the overhead of SSL processing from the HTTPS web server transaction. 2) Within SSL processing, isolate the overhead of handshake phase from the bulk data transfer phase and study the major components in each phase. 3) Analyze the crypto operations in the SSL processing in terms of their architectural characteristics like path length, cycles per instruction and frequently used instructions.

To achieve these three major goals, we have the following three setups for our experiments.

3.1. SSL Analysis in Web Servers

The web server we use is a 2.26GHz Intel® Pentium® IV based workstation with 512MB of memory. The client machine is a DP system with Intel® Xeon™ processor running at 2.6GHz and with 1GB of memory. Both machines run Linux 2.6.6. The Apache 2.0 [1] server together with mod_ssl (which is an interface to the OpenSSL library) is used as the web server. The client software based on curl [4] generates multiple secure HTTP requests. Both the server and the client machine are installed with OpenSSL 0.9.7d [16] as the SSL library, which is compiled using the optimizations for Pentium processor. The client makes HTTP requests as fast as the server can handle them. During our experiments, the server load is always maintained at more than 90%. The tools we use to perform the measurements are Oprofile [17], which is a system-wide profiler for Linux systems. By profiling all the running code at a low cost, Oprofile enables us to identify the time spent in various modules and functions.

OpenSSL supports SSL v2/v3 and TLS v1 protocols as well as a cryptography library. Our experiments employ the widely used SSL v3. The cipher suite we use is DES-CBC3-SHA, where (1) the RSA algorithm is used for signing as well as the public key encryption, (2) 3DES in CBC mode is used as the private key encryption, and (3) SHA-1 is used as the hashing function for MAC calculation. MD5 is also used in the handshake phase for calculation of hash values in the finish messages.

3.2. Standalone SSL Setup for Detailed Analysis

To focus on the SSL processing itself without including any networking or IO overhead, we use a standalone program running on the same server machine as the second setup. We modify the program *ssltest* for this purpose. This program creates a server context as well as a client context, and relays messages between these two through some memory buffers. Our measurements are taken on the server side. The same OpenSSL library and the same cipher suite are used as well. To get the latency on those components that we are interested in, we use the read timestamp instruction.

3.3. Standalone Crypto Benchmark

The crypto operations are the main components in the SSL protocol processing. To study these operations, we developed a crypto benchmark, which essentially makes various function calls into the crypto library that comes with the SSL library. We use Vtune [9] and SoftSDV [19] to do profiling and tracing the instructions executed respectively. The Vtune analyzer allows time-based sampling as well as event-based sampling at various levels – from processes to modules to functions. SoftSDV is a full system simulation environment. It provides a virtual platform that consists of a simulated CPU and other platform components. SoftSDV allows an actual Operating system to be loaded so that the crypto benchmark can be executed within it. The instruction traces collected from SoftSDV are then analyzed through various simulation tools to understand the execution profile.

4. Analysis of SSL Processing

In this section, we focus on studying the anatomy of the important SSL phase(s). We first look at the execution time breakdown in SSL processing in a web server. We then study in detail how the SSL handshake is performed on the server side and how the execution time is distributed across the various steps.

4.1 Execution Time Breakdown in SSL

Table 1 shows the execution time breakdown on various components when the web server processes a 1KB web page. Similar results can be obtained when we increase the request file size. We can see that the SSL processing (libssl and libcrypto) takes 71.6% of the total processing time, which is mostly due to the crypto operations (libcrypto). We further breakdown the crypto operations into four components: public key encryption, private key encryption, hashing and other

operations (including random number generation, etc). Figure 2 shows the execution time breakdown in the crypto library as we vary the request file size. It can be seen that the public key encryption takes a very large portion: about 90% when the request file size is 1k bytes. This portion reduces as we increase the file size. Since the public key encryption is one of the main operations in the SSL handshake, it makes the handshake quite expensive. Session re-negotiation using the previously setup keys can avoid the public key encryption, therefore greatly reduces the handshake overhead. On the other hand, the portion of the private key encryption and hashing are increasing as we increase the request file size. This is obvious since the cost of encryption and hashing is proportional to the request file size. The private key encryption portion is negligible with a small file size. It is only 2.4% when the file size is 1K bytes. However, as this part is the main contributor to the bulk data transfer, it can become significant at very large file size. Therefore for workloads that have large request file size or long sessions of data exchange (e.g. B2B sessions), optimizations should be concentrated on both private key encryption and public key encryption.

Table 1. Execution time breakdown in web server

Components	Functionality	%
libcrypto	Crypto library, including all the cryptography functions	70.83
libssl	SSL functions	0.82
httpd	Apache web server	1.84
vmlinux	Linux kernel, including TCP stack processing	17.51
other	Other library, including c, thread library, etc.	9.00

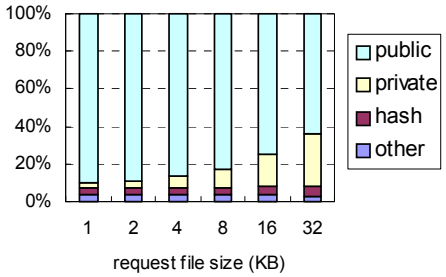

Figure 2. Time breakdown in crypto library

4.2 Anatomy of SSL Handshake

The SSL handshake on the server side can be partitioned into 10 steps based on its functionality. For each step, we measure the total processing time as well as the time spent on the crypto operations. Table 2 shows the results in complete detail.

The server receives and processes the client hello message in step 1 after its initialization for internal data structures. In step 2, the server generates a random number, session id, etc. and sends out the server hello message. Then the server's certificate followed by a server done message are sent in step 3 and 4 respectively. In the cipher suite we use for the experiments, the certificate contains the RSA public key for key exchange, therefore the server key exchange message is skipped.

In step 5, the server receives the client key exchange message that contains a 48-byte key called pre-master. It is encrypted using the server's RSA public key in the client side. The server decrypts this pre-master using its private key. This pre-master is used to generate another key called master through a series of hash functions (both MD5 and SHA-1 are used).

In step 6, the server receives the change cipher spec message from the client, which indicates that the subsequent message will be encrypted by the finally generated private keys. At this moment, the server calculates the key blocks using the master key generated in the previous step. This calculation is also a series of hash functions that are similar to the master key generation. These keys are used for private key encryption as well as MAC calculation, which are performed in the bulk data transfer. Upon this point, the server has received all the handshake messages from the client, and also transmitted its own handshake messages to the client. (Handshake messages include all the received and transmitted messages except change cipher spec message.) The server then calculates two finish hash values (MD5 and SHA-1 respectively) over these handshake messages for the client side using 'CLNT' padding. After this is done, it reads the client finished message. Note that this is the first message that uses the private key encryption. Therefore, the server decrypts this message using the private key. The MAC together with this message is also calculated to ensure its integrity. After retrieving the two finish hash values inside the finish message, the server verifies that they are same as the ones that are just calculated.

In step 7, the server sends out its own change spec cipher message. Finally in step 8, the server calculates the two finish hash values with 'SRVR' padding. These two values together with the MAC value for this message are encrypted using the private keys, and are sent out as the server finish message. Then the server flushes its internal memory and cleans data structures that will not be used (like the pre-master and master keys). All the subsequent messages will be en/decrypted using the private keys.

Since the finish hash values are calculated based on all the handshake messages that are received as well as transmitted, and because the hash value is calculated

Table 2. Execution time breakdown in SSL handshake

Step	Functionality	Descriptions	Latency (1000s of cycles)	Crypto Functions Called	Latency (1000s of cycles)
0	Init	Initialize states and variables	348	init_finished_mac	29
1	get_client_hello	check version, get client random, session-id and generate new session id check if compression is needed, choose a cipher from the cipher list	198	rand_pseudo_bytes finish_mac	68 1.4
2	send_server_hello	generate server random, send server hello message	61	rand_pseudo_bytes finish_mac	40 5.8
3	send_server_cert	Send server certificate	239	X509 functions finish_mac	232 16
	skip server_kx		0.6		
	skip cert_req		0.1		
4	send_server_done	Send server done message	3.8	finish_mac	1.65
	server_flush	Internal buffer control	3.4	BIO_ctrl, BIO_flush	3.4
5	check_client_hello	Read client_kx message	12	finish_mac	5.6
	get_client_kx	get pre-master using rsa-private-decryption, generate master key from it	18941	rsa_private_decryption , gen_master_secret, cert_verify_mac	18563 148 61
6	get_cert_verify	a. read client change cipher spec, generate key block from master key, calculate hash values for finish message b. read client finished message, decrypt it, calculate mac	293	a. gen_key_block final_finish_mac b. pri_decryption mac finish_mac	106 62 16 33 3.1
	get_finished	compare the finish hash values in the client finished message with the previously computed one	0.74		
7	send_cipher_spec	Send server change cipher spec message	38		
8	send_finished	calculate server finish hash values for finish message , Calculate mac, encrypt it	114	finial_finish_mac mac pri_encrytpion, finish_mac	64 31 11.5 3.5
9	server_flush	Internal buffer control	2.5	BIO_ctrl, BIO_flush	2.5
	check state; end		287		
	Total		20540		19506

based on a block of 64 bytes, in OpenSSL's implementation, the two hash values are calculated whenever a handshake message is received or transmitted. They are finalized when the last handshake message is received. This is why the hashing functions are called in most of the steps.

Table 3. Crypto operations during SSL handshake

Functionality	Latency (Cycles)	%
Public key encryption	18562720	90.4
Private key encryption	27260	0.1
Hash functions	569948	2.8
Other functions	346354	1.7
Total crypto operations	19506282	95.0
Total SSL processing	20540392	100

We can see from Table 2 that the main crypto operations used in SSL handshake are public key encryption, hashing and private key encryption. Table 3 summarizes the total time spent on these crypto operations during the handshake process. Public key encryption takes about 90.4% and hashing takes 2.8%. Since the private key encryption is performed on only two messages, this part is almost ignorable. Other functions include random number generation, X509 functions for server certificate, etc., which also take a very small portion. In total, the crypto operations take about 95.0% of the SSL handshake.

5. Anatomy of Crypto Operations in SSL

We have shown that crypto operations are the main bottleneck in SSL processing. In this section, we look in detail at these operations.

5.1. Symmetric Key Encryption

We study two block ciphers -- AES and DES/3DES, and a stream cipher -- RC4. Since the symmetric key cryptography has a similar process for both encryption and decryption, we only talk about encryption here. The

encryption process consists of two phases: a key setup followed by an encryption kernel. The key setup is to initialize a key schedule (for block ciphers) or a state table (for stream ciphers) based on the input private key. Both the key schedule and the state table are in the form of an array. Data in this array are accessed during the encryption kernel. Figure 3 shows the portion of key setup in the encryption process as we vary the encryption data size. While this portion is quite small for the block ciphers (only 1.0% ~ 3.6% even when the transferred data size is 1Kbytes), it is much higher for RC4 (28.5% for a 1Kbytes of data). This is because RC4 has a much simpler encryption kernel than AES and DES/3DES, and because the key setup for RC4 is to initialize a bigger table which has 256 entries. AES and DES/3DES use a smaller sized key schedule. In any case, however, the key setup portion is decreasing as we increase the data size. When the data size is increased to 8K bytes, this portion becomes less than 0.5% and 5% for AES/DES/3DES and RC4 respectively. At an even larger data size, this portion is almost negligible.

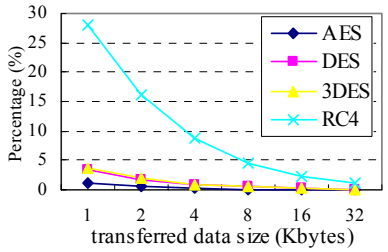

Figure 3. Key setup during encryption

Table 4. Important data structures and characteristics
(* key size for AES can be 192 or 256 bits)

	AES	**DES**	**3DES**	***RC4***
Block Size	128b	64b	64b	8b
Key Size	128b*	56b	3x56b	128b
Key Schedule	44,32b	32,32b	3x(32,32b)	n/a
Tables	4,256,32b	8,64,32b	8,64,32b	1,256,8b
Rounds	10	16	3x16	1
Table Lookup	16	8	8	3

After initialization, the encryption kernel consists of a series of block operations depending on the input data size. Since RC4 is not a block cipher, its encryption is performed on a unit of one byte. So when we talk about block operations in general, it is referring to a one byte operation for RC4. Table 4 shows the main data structure and characteristics for each block operation. All the tables except for RC4 contain constant values that are initialized as static data structures. For AES and DES/3DES, each block operation consists of several iterations (rounds) of basic operations, which are essentially logical operations and table lookups. The number of table lookups (not including accessing the key schedule array) for each round is listed in the table as well. For RC4, the generation of pseudo-number also includes 3 table lookups. In the subsequent subsections, we will look at how these block operations are performed and what are their architectural characteristics.

1) AES Performance Breakdown

The block operation on AES can be divided into the following 3 parts: 1) Map byte array block to cipher state and add initial round key. Its main operations are shift and XOR. 2) Main rounds (9 for 128 bit key and 13 for 256 bit key) of byte substitution, row shift transformation, column transformation and round key addition are performed. The main operations are table lookup, shift and XOR. 3) The last round and map the cipher state to byte array. Its main operations are same as those in step 2. Table 5 list the latency for each part on the block operations with a 128 bit key and a 256 bit key. We can see that the main rounds take a large portion. It is about 70.64% and 77.91% for a 128-bit key and 256-bit key respectively. Larger key size only affects the second part since the first and the last parts are fixed.

Table 5. AES execution time breakdown

Step	**Functionality**	**128 bit key**		**256 bit key**	
		Cycles	**%**	**Cycles**	**%**
1	Map byte array block to cipher state, add initial round key	69	12	69	9
2	Main rounds	397	71	582	78
3	Last round and Map cipher state to byte array	96	17	96	13
	Total	562	100	747	100

One round consists of four basic operations. Each basic operation takes four 32-bit inputs and generates one 32-bit output. One byte is taken from each of the four inputs, and is indexed to one of the four tables. The four values from the tables are XORed together with the corresponding key value from the key schedule to generate one output. Each basic operation takes a different byte from the input and index to different tables. The byte used to index the table is taken in the order of left rotate within one input. Four such operations generate four outputs in one round. The four outputs become the four inputs for the next round.

2) DES / 3DES Performance Breakdown

The block operation on DES/3DES can be divided into the following 3 parts:

(1) Initial permutation -- Its main operations are shift, AND, XOR, and rotate.

(2) Substitution -- This part consists of one and three sets of 16 rounds for DES and 3DES respectively. The main operations are XOR, rotate, AND, shift and table

lookup.

(3) Final Permutation -- This part has similar operations as the first part.

Table 6. DES/3DES execution time breakdown

Step	Functionality	DES		3DES	
		Cycles	%	Cycles	%
1	IP	50	13.15	55	5.3
2	Substitution	286	74.74	915	89.1
3	FP	46	12.11	57	5.6
Total		382	100	1027	100

We measured the latency spent on each part for DES and 3DES. As shown in Table 6, the second part is the main bottleneck. It is 74.7% and 89.1% for DES and 3DES respectively. In each round of the substitution part, it takes two 32-bit inputs and generates two outputs. The basic operation is also table lookups. First two 32-bit temporal data are obtained by XOR operations between one of the inputs and its corresponding key value from the key schedule. Then all the eight bytes from these two temporal data are indexed to the eight tables. Each byte is shifted 2 bit right so that only six bits are used as the index. Finally the eight values from the eight tables are XORed with another input and write it back. The two outputs are exchanged as inputs for the next round.

3) RC4 Performance Summary

RC4 is fairly simple compared to the previous crypto operations. Essentially it uses a pseudo-random generation algorithm to generate a byte stream, which is XORed with the input stream. We did not breakdown the execution time further because the encryption routine itself is a single step. During the generation for each of the pseudo-random numbers, the state table with 256 entries is read 3 times and updated twice. The main operations are AND, ADD and XOR.

5.2. Asymmetric Key Encryption

As the web server performs RSA decryption on the client key exchange message, we focus on the RSA decryption process. RSA decryption can be partitioned into six parts:

(1) Initialization -- The internal data structures and memory buffers are initialized.

(2) String to Big Number conversion -- The input of the RSA decryption in SSL implementation is an octet string, which needs to be converted into a multi-precision integer.

(3) Blinding -- This is used to avoid a time attack [3].

(4) RSA computation -- This is where the real computation is performed using the private key. The main operations in this part as well as the previous part are exponentiation and modulus. The output for this part is also a multi-precision integer.

(5) Big number to octet string conversion -- This is the reverse of step 2.

(6) Block parsing -- This part is required because before the client uses the server's public key to encrypt the plaintext, the plaintext is first padded into a string that has some format and length defined in PKCS #1 [18]. Therefore, to recover the plaintext, the reverse operation is performed.

Table 7. Execution time breakdown for RSA

Step	Functionality	512b		1024b	
		Cycles	%	Cycles	%
1	Init	866	0.07	936	0.02
2	data_to_bn	783	0.07	1189	0.02
3	blinding	14319	1.20	39783	0.66
4	computation	1159628	97.01	5972288	98.85
5	bn_to_data	587	0.05	1053	0.02
6	block_parsing	19107	1.60	26104	0.43
Total		1195290	100	6041353	100

Table 7 lists the time breakdown for each step with a 512-bit key and a 1024-bit key respectively. As expected, the RSA computation is the main bottleneck. It takes about 97.0% and 98.8% of the total processing for the two keys. The main reason is due to the extensive computation for multi-precision integer operations such as multiplication and addition.

Table 8. Top Ten Functions in RSA

Function	%
bn_mul_add_words	47.04
bn_sub_words	22.61
BN_from_montgomery	9.47
bn_add_words	4.92
BN_usub	3.24
BN_copy	1.50
ERR_load_BN_strings	1.77
OPENSSL_cleanse	1.59
BN_sqr	1.04
BN_CTX_start	0.77

Table 9. Instructions in bn_mul_add_words()

```
movl  0x8(%ebx), %eax
mull  %ebp
addl  %esi, %eax
movl  0x8(%edi), %esi
adcl  $0x0, %edx
addl  %esi, %eax
adcl  $0x0, %edx
movl  %eax, 0x8(%edi)
movl  %edx, %esi
```

There are many function calls in RSA decryption process. We list the top ten functions in Table 8 based on the total time spent on them. This data is collected with a 1024 bits key. We can see that the function bn_mul_add_words() is most time-consuming. It takes about 47.0% of all the processing time. This function is in fact very small and simple. Its basic operation is to do a multiplication followed by two additions. Table 9 lists its corresponding instructions that are executed. It can be seen that MULL, ADDL, ADCL (add with carry) are the main instructions used.

5.3. Hash Functions

Both MD5 and SHA-1 add padding to ensure that the padded message is a multiple of 512 bits. Then the message is parsed using a sequence of logical functions at a block size of 64 bytes. The hashing processes can be partitioned into 3 parts:

(1) Initialization -- The internal states are initialized. SHA-1 has more states than MD5.

(2) Update -- The hashing functions are performed on the input data in a block size of 64 bytes. The block operation and the number of block operations are different for MD5 and SHA-1, with SHA-1 more compute intensive.

(3) Final -- One or two block operations are performed on what is left from the previous step (some padding are added), and generates the final signature.

Table 10. Execution time breakdown for MD5 and SHA-1

Step	Functionality	MD5		SHA-1	
		Cycles	%	Cycles	%
1	Init	59	0.88	66	0.62
2	Update	6070	90.88	9871	92.05
3	Final	550	8.24	786	7.33
Total		6679	100	10723	100

We take 1024 bytes data as the input, and measure how much latency spent on each part. As shown in Table 10, the second part is most time-consuming. This is obvious because the main block operations are performed in this part. The block operation essentially contains a lot of logical operations such as XOR, and shift operations.

6. Architectural Characteristics and Optimizations

In this section, we summarize the crypto operations by showing their architectural characteristics like instruction distribution, CPI and throughput they can achieve. We then talk about some optimizations and inferences that can improve the performance of crypto operations, which in turn improve the SSL processing.

6.1. Architectural characteristics

Table 11 shows the CPI, path length and throughput for these crypto operations. Since all of these crypto operations are compute intensive, their CPI is very low (0.52 to 0.77). RSA has the highest CPI due to its multiplication operations. From the path length (instructions per byte), we can see the complexity of these crypto operations. RSA has the longest path length. MD5 and SHA-1 have the shortest one. And the path length for private key encryption is between the previous two. In private key encryption, 3DES is the most complex one. The throughput is mostly determined by the path length since they have the similar CPI. Again, RSA can achieve the throughput at only 0.036 Mbytes/s. Hashing functions are much faster, with MD5 even faster than SHA-1. For the private key encryption, it can achieve a throughput of about 13.32 to 211.34 Mbytes/s (which corresponds to 106 Mbps to 1.7 Gbps), with 3DES and RC4 the lowest and the highest respectively. Although AES is faster than 3DES, it is still incapable of saturating a network link running at 1Gbps.

Table 11. Characteristics for crypto operations

Crypto Operations	Private key				Public key	Hashing	
	AES	DES	3DES	RC4	RSA	MD5	SHA-1
CPI	0.66	0.67	0.66	0.57	0.77	0.72	0.52
Path length (Instructions per byte)	50	69	194	14	61457	12	24
Throughput (MB/s)	51.19	36.95	13.32	211.34	0.036	197.86	135.30

Table 12 shows the top ten instructions that are frequently used for these crypto operations. These instructions take 89.78% to 98.53% of the total instructions that are executed. We can see that the move instruction, which load/store data from/to the memory, is the top one instruction for all these operations except DES/3DES. This is because there is very limited number of registers in Intel's x86 architecture. Since all these crypto operations are compute intensive, most of these move instructions are hits in the L1 cache. However, adding more general purpose registers to the CPU may help reduce the number of these move instructions, which in turn will reduce the execution time. For private key encryption and hashing functions, logical operations like XOR and AND are the frequently used instructions as expected. Shifts or rotates also take a large portion (except for RC4). For RSA, the compute instruction like ADD and ADC (add with carry) is very frequently used, followed by the multiply instruction. These instructions are mostly used in the exponentiation and modulus operations.

6.2 Optimizations and Inferences

To accelerate the SSL processing, many techniques can be employed to improve the crypto operations. We classify the potential techniques into three categories: 1) ISA support, 2) Hardware units and 3) crypto engines. It should be noted that a detailed evaluation of these optimizations is not within the scope of this paper. However, based on our characterization, we believe that the following are the important types of optimizations in above mentioned categories:

(1) ISA support for secure processing includes adding

Table 12. Top ten instructions for crypto operations

Private key								Public key		Hashing			
AES		DES		3DES		RC4		RSA		MD5		SHA-1	
movl	37.75	xorl	41.11	xorl	39.80	movl	38.06	movl	37.17	Movl	22.11	movl	27.81
xorl	25.09	movb	17.54	movb	18.76	andl	18.15	addl	16.25	Addl	19.12	xorl	22.40
movb	11.52	movl	13.54	movl	13.49	addl	13.61	adcl	16.18	Xorl	18.58	addl	12.04
andl	7.40	andl	13.52	andl	13.16	movb	6.35	mull	6.10	Leal	9.15	roll	10.14
shrl	4.11	shrl	5.85	shrl	6.25	incl	6.18	pushl	4.81	Roll	8.88	leal	5.77
decl	2.26	rorl	3.29	Rorl	3.71	nop	5.96	popl	2.44	Andl	4.75	rorl	5.64
jnz	2.16	roll	1.83	Roll	1.11	xorl	1.82	jnz	2.24	Movb	4.24	andl	4.39
incl	1.65	pushl	0.75	pushl	1.05	cmpl	1.43	subl	1.95	Orl	2.31	orl	2.86
xorb	1.65	popl	0.74	popl	1.04	popl	1.13	xorl	1.34	Addb	1.57	movb	2.25
pushl	0.93	addl	0.37	Ret	0.26	pushl	1.08	cmpl	1.29	Pushl	1.21	bswap	1.06
Total	94.52		98.53		98.63		93.75		89.78		91.91		94.36

new instructions to replace a series of instructions for a basic operation. For instance, MD5 and SHA-1 use a lot of logical operations, which take three inputs and generate one output. Figure 4 shows some of the examples. All the operations are simple functions like AND and XOR. However since logical instructions take only two operands, these operations take at least two instructions like (b). For (a) and other operations, many more instructions are needed. In addition, since these operations can only employ the limited set of x86 registers, this series of computations also introduces several move operations to save the data to memory and fetch it back. To avoid these overheads, a single instruction that allows for three operands can replace these series of operations. Another approach is to continue to use two operands but require larger registers (like MMX registers). One 128-bit MMX register can contain multiple 32-bit or 64-bit data and therefore can implicitly represent more than two operands.

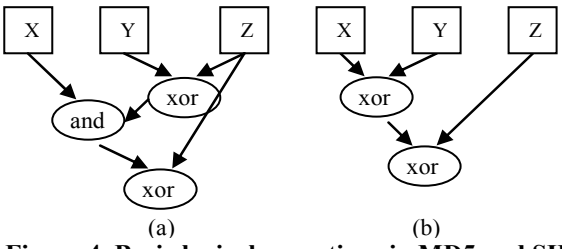

Figure 4. Basic logical operations in MD5 and SHA1

(2) Hardware support can be added to perform some crypto operations at a higher level. For instance, there are a lot of table lookup operations in AES and DES/3DES. Figure 5 shows hardware support that can be used to perform one round in AES algorithm. S0 to S3 and T0 to T3 are the four inputs and outputs respectively. Te0 to Te3 are the four tables that contain constant values. KS is the key schedule. As described in the previous section, each round consists of four basic operations. As shown in the figure, in the first basic operation (in solid lines indicated by '0'), the first byte from S0 is indexed to table Te0, the second byte from S1 is indexed to table Te1, and so on. The four values from the four tables are XORed with the corresponding key value from the key schedule. This results in the first output (T0) for this round. In the next basic operation (in dashed lines indicated by '1'), the fourth byte from S0 is indexed to table Te3, the first byte from S1 is indexed to table Te0, and so on. The next two outputs are generated in a similar way. Note that these four basic operations have no dependency on each other, therefore can be performed in parallel completely. To drive this hardware unit, a new instruction needs to be added after assigning the four input registers. Since T0 ~ T3 become the inputs for the next round, this hardware unit can be extended to perform all rounds and return the final four outputs.

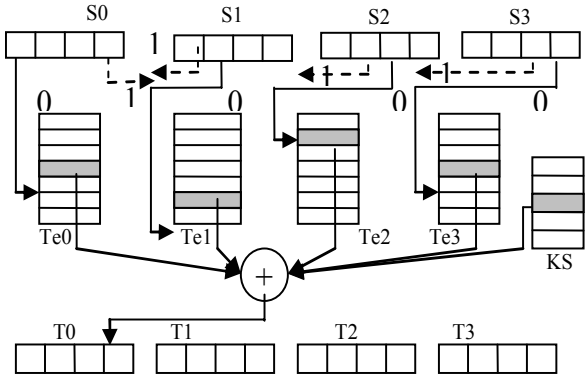

Figure 5. Hardware support for AES table lookup

3) Crypto engines can be used at an even higher level than the hardware unit. It can support a complete encryption algorithm. For instance, a crypto engine that supports AES can take a string as its input and

generates the encrypted data as the output. In addition, crypto engines can run asynchronously with the CPU, so that the throughput can be improved significantly. Furthermore, several crypto units within one engine can run in parallel in the bulk transfer phase. For instance, when the web server tries to send a 1K bytes data to the client using AES encryption, the data actually sent is an encrypted fragment that consists of the 1K bytes data, the MAC value and some padding (to make the fragment a multiple of the block length). What the server does is that it first calculates the MAC of the data, and then performs AES encryption on the fragment. With the support of a crypto engine that includes an AES encryption unit and a hashing unit, the AES encryption and the MAC calculation can be performed in parallel. As shown in Figure 6, the control unit in the engine fetches data from the memory (via reading descriptors set by the user program) and feed it into the AES encryption unit as well as the hashing unit. The AES encryption unit generates the first part of the fragment. When the hashing unit finishes the MAC calculation, the MAC value and the padding are feed into the AES encryption unit and generate the last part of the fragment.

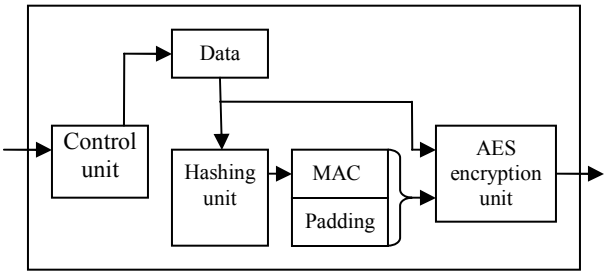

Figure 6. Pipelining/parallelism in crypto engines

7. Conclusions and Future Work

In this paper, we analyzed the SSL performance in secure web transactions. It turns out that about 70% of the total processing time of an HTTPS transaction is spent in SSL processing. As a result, a more detailed understanding of the key overheads within SSL processing was required. By presenting a detailed description of the anatomy of SSL processing, we showed that the major overhead incurred during SSL processing lies in the session negotiation phase when small amount of data are transferred (as in banking transactions). On the other hand, when the data exchanged in the session crosses over 32K bytes, the bulk data encryption phase becomes important. We then showed the breakdown of time spent on the cryptographic operations that were classified as asymmetric encryption algorithms, symmetric encryption algorithms and hash functions.

Our final contribution was a more detailed analysis of the commonly used crypto algorithms to determine the time consuming operations (table lookups, permutations, logical operations, etc) occupies a significant fraction of the execution time. We presented the architectural characteristics of crypto operations by analyzing CPI, path length and frequently used instructions. Finally we presented our inferences on ISA/hardware support to improve the SSL processing. Our future work involves investigating the design and performance of architectural support for security protocols further.

References

[1] Apache, HTTP server project, http://httpd.apache.org/

[2] Jerome Burke, John McDonald, Todd Austin, "Architectural Support for Fast Symmetric-Key Cryptography", Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS) , pp. 178-189, Nov. 2000

[3] D. Brumley and D. Boneh, "Remote Timing Attacks are Practical", USENIX, 2003

[4] Curl, http://curl.haxx.se/

[5] T. Dierks, C. Allen, The TLS Protocol Version 1.0, http://www.ietf.org/rfc/rfc2246.txt

[6] Whitfield Diffie, Martin E. Hellman, "New Directions in Cryptography", IEEE Trans. Info. Th. 22, 644-654, 1976

[7] A.O. Freier, P. Karlton, P.C. Kocher, "The SSL Protocol, V3.0", IETF draft, http://wp.netscape.com/eng/ssl3/3-spec.htm

[8] Intel, "IXP2850 Network Processor", http://www.intel.com/design/network/products/npfamily/ixp2850.htm

[9] Intel, "VTune Performance Analyzers", http://intel.com/sfotware/products/vtune

[10] Krishna Kant, Ravi Iyer and Prasant Mahapatra, " Architectural Impact of Secure Socket Layer on Internet Servers", Proc. Int. Conf. on Computer Design, Sep. 2000

[11] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, Nov 1998

[12] Ruby B. Lee, Zhijie Shi and Xiao Yang, "Efficient Permutation Instructions for Fast Software Cryptography", IEEE Micro, Vol. 21, No. 6, pp. 56-69, December 2001

[13] A.J. Menezes, P.C. Van Oorschot, et al., "Handbook of Applied Cryptography", CRC Press, Oct. 1996

[14] NIST, Advanced Encryption Standard (AES), FIPS Pub. 197, http://csrc.nist.gov/publications/fips, Nov, 2001

[15] NIST, Secure Hash Standard (SHS), FIPS Pub. 180-2, http://csrc.nist.gov/publications/fips, Aug. 2002

[16] OpenSSL, http://www.openssl.org/

[17] Oprofile, http://oprofile.sourceforge.net/news/

[18] PKCS #1: RSA Cryptography Standard, http://www.rsasecurity.com/rsalabs

[19] R. Uhlig, R. Fishtein, et al., "SoftSDV: A Pre-silicon Software Development Environment for the IA-64 architecture", Intel Technology Journal, 4th quarter, 1999

[20] Lisa Wu, Chris Weaver, and Todd Austin, "CryptoManiac: A Fast Flexible Architecture for Secure Communication", Int. Symposium on Computer Architecture (ISCA), 2001