Encrypting the Internet

Michael E. Kounavis
Intel Architecture Group
2111 NE 25th Avenue
Hillsboro, OR 97124
michael.e.kounavis@intel.com

Xiaozhu Kang
Intel Labs
2111 NE 25th Avenue
Hillsboro, OR 97124
xiaozhu.kang@intel.com

Ken Grewal
Intel Labs
2111 NE 25th Avenue
Hillsboro, OR 97124
ken.grewal@intel.com

Mathew Eszenyi
Intel Labs
2111 NE 25th Avenue
Hillsboro, OR 97124
mathew.eszenyi@intel.com

Shay Gueron
Intel Architecture Group
Israel Development Center
Haifa, Israel
shay.gueron@intel.com

David Durham
Intel Labs
2111 NE 25th Avenue
Hillsboro, OR 97124
david.durham@intel.com

ABSTRACT

End-to-end communication encryption is considered necessary for protecting the privacy of user data in the Internet. Only a small fraction of all Internet traffic, however, is protected today. The primary reason for this neglect is economic, mainly security protocol speed and cost. In this paper we argue that recent advances in the implementation of cryptographic algorithms can make general purpose processors capable of encrypting packets at line rates. This implies that the Internet can be gradually transformed to an information delivery infrastructure where all traffic is encrypted and authenticated. We justify our claim by presenting technologies that accelerate end-to-end encryption and authentication by a factor of 6 and a high performance TLS 1.2 protocol implementation that takes advantage of these innovations. Our implementation is available in the public domain for experimentation.

Keywords

Secure Communications, Cryptographic Algorithm Acceleration, SSL, TLS, HTTPS, RSA, AES, GCM

1. INTRODUCTION

Today there are more than 50 million web sites in the Internet [13]. However, only about 600,000 of those offer SSL security [7]. We believe that end-to-end Internet encryption and authentication is important for many different reasons. First people value their privacy, feeling uneasy when someone listens on to their conversations or views what they access in the Internet. Second, Internet services are prone to numerous types of attacks such as phishing, virus, and worm attacks resulting in identity thefts, denial of service and data breaches. The need for secure channel communication across the Internet becomes even more important due to the ever increasing network traffic and number of online transactions.

Despite the obvious need for end-to-end security in the Internet, most of the traffic is not encrypted or protected today. The primary reasons for this are protocol speed and cost. Cryptographic algorithms consume millions of clocks to execute on general purpose processors. Alternative approaches such as the use of specialized appliances work but they are much more expensive [12] than doing the SSL/TLS termination using general purpose hardware.

In this paper we address this problem by investigating technologies that accelerate cryptography by factors. We tackle this important networking problem using novel circuit technologies and mathematical tools. First, we present a set of processor instructions [6] that accelerate the rounds of the Advanced Encryption Standard (AES) resulting in an overall acceleration gain of about 12x as compared to other software solutions, for many modes of AES. Second we present a set of software optimizations to implementations of the Rivest Shamir Adleman (RSA) public key encryption algorithm that offer an additional 40% gain on the performance of the public key portion of Internet transactions. Third, we discuss how message authentication can be supported by the Galois Counter Mode (GCM) mode of AES much more efficiently than other alternatives. Our GCM implementation is based on another new processor instruction [6] for doing carry-less multiplication and a new algorithm for doing reduction in binary finite fields.

Combining our technologies and general purpose hardware we build a TLS 1.2 server capable of supporting approximately 1200 banking transactions per second per processor core, resulting in a 6x overall protocol performance gain. The work entailed a protocol implementation change, a proper cryptographic suite selection and hardware acceleration. Moreover we show that only 8 processor cores running at 3 GHz are needed to saturate a 10 Gbps link with TLS 1.2 traffic. Supporting 8 cores by general purpose hardware can be achieved today.

The paper is structured as follows: In Section 2 we discuss the motivation behind our project and research. In Section 3 we present related work. In Section 4 we present an overview of the SSL/TLS protocols and the AES and RSA crypto algorithms, which we accelerate. The details of our crypto acceleration technologies are presented in Section 5. Section 6 discusses the development of a high performance TLS 1.2 protocol implementation that leverages our technologies and Section 7 presents our performance results. Finally in Section 8 we provide some concluding remarks.

2. MOTIVATION

The landscape of security today is that only traffic carrying sensitive information such as banking and Ecommerce is encrypted. In fact, most of the network traffic in our everyday communication (e.g., e-mail, web surfing, instant messaging, social networking and peer-to-peer applications) is sent in the clear. We believe that end-to-end encryption and authentication is important because people are sensitive about their security and privacy.[1] For example, people have the expectation that nobody else is listening to their conversation when they have a phone call. Likewise people have the expectation no one is eavesdropping on their Internet traffic. Unfortunately, viewing Internet traffic is fairly easy today.

It is equally easy to mislead the user to visit malicious web sites. End users are typically not cognizant of the distinction between signed and unsigned URLs. If every URL is signed then users can be confident that they are going to the web sites they see in their URL. Because of this reason, phishing attacks can be greatly reduced when TLS authentication controls are always in place. Phishing is also one method for enabling attacks caused by viruses, worms and rootkits. Viruses and worms usually spread through e-mail. With the right cryptographic mechanisms in place, the sources of viruses and worms can be forced to expose parts of their identity (e.g., through e-mail signing or DNSSEC) and hence become accountable. This is a first step toward reducing virus and worm attacks. The study in reference [44] analyzes virus-caused data breaches experienced by 43 U.S. companies in 17 different industry sectors. It is found that an average breach costs each company \$6.6 million in 2008 [4, 39, 44].

Alarmingly, most of the HTTP traffic is still not encrypted or protected [20, 43, 47]. Less than 10% of the world's web sites are HTTPS enabled today [7, 13]. This is primarily due to the cost associated with providing this additional security as well as the performance penalty overhead in providing the additional cryptographic operations. It is perceived that there is a tradeoff between security and performance [11]. One thing in common between the various cryptographic algorithms used today is the complexity and cost associated with their implementation. For example encrypting the content of a typical 140 KB banking transaction requires 2.3-4.8 million clocks, depending on the implementation on a state-of-the-art 3 GHz Intel® Core™ i7 processor core. Similarly, performing an RSA 1024 private decrypt operation requires 0.9-1.4 million clocks on the same processor. The total worst case cost related to cryptography, including the SHA-1 overhead, for processing a 140 KB banking transaction is 7.3 million clocks. These numbers mean that it is cost prohibitive for general purpose processors to be used for protecting large amounts of HTTPS traffic at high link speeds. Today dedicated appliances [2], supporting about 10K concurrent connections, are used for providing end-to-end security. Solutions employ dedicated hardware accelerators and front end terminators for TLS connections.

In this paper we present an alternative methodology based on using general purpose processors for achieving similar performance. It should be understood that we are not solving the only problem related to Internet encryption. There are other issues besides protocol speed and cost which need to be tackled such as the generation, distribution and management of certificates. User privacy may also be violated at the back-end server, even if the traffic is encrypted. Employing end-to-end security protocols such as TLS also prevents existing tools such as intrusion detection/prevention systems and traffic shapers from performing much needed network functions. To address this problem, companion technologies such as [24] can be used for enabling authorized access to encrypted data.

3. RELATED WORK

Our work encompasses many aspects of cryptographic algorithm acceleration [41, 42, 38, 27, 21, 36, 17, 29, 30, 5]. In what follows we summarize some representative pieces of work and explain how our paper differentiates from them. In the area of AES acceleration, several AES processing units have been developed in the past [26, 45, 27]. The main difference between our work and references [26, 45, 27] is that in our work AES is broken down to its fundamental components (i.e., the Galois Field processing done by the round transformations) and exposed as a set of instructions to the programmer.

The concept of instructions for accelerating the rounds of AES, which we use in our design, is also described in references [23, 19]. The circuits of these references are based on table lookups or more generic Galois computations and hence may consume larger area and power than ours. Another aspect of our work is that we use composite field technologies to reduce the area requirement of AES. Several composite field implementations of AES are reported in [41, 42, 38, 48, 18, 35]. These circuits however are mostly reported in the context of ASIC AES implementations. What makes our work different from these pieces of work is that we are among the first to combine the concept of an AES round instruction with the use of composite field technologies for reducing the area requirement. In this way we not only provide flexibility to the programmer, but also reduce the area requirement of the AES logic to such a degree that it can fit inside general purpose processor cores (see Section 5).

In the area of RSA software acceleration past research has focused on the modular multiplication aspect of it (i.e., the Montgomery [36, 30] or Barrett [16] algorithms), on the integer multiplication component [28, 46, 37, 31] or the windowing technique used [29]. Efficient software RSA implementations can be found in [8, 3]. Our work is different from these references in that the Montgomery algorithm is reduced to 1.5 big number multiplications which are implemented very efficiently using a register recycling technique. Our work is also different from [46, 31] since we use a more efficient set of integer multiplication routines.

Finally, in the area of message authentication, we propose a new GCM implementation which is much faster than other schemes based on HMAC-SHA1 [34] and prior art for implementing GCM [5] based on performing table lookups. In our approach GCM is implemented by splitting the field-dependent from the field-independent part of the algorithm

[1]Typically outside of national security concerns.

and performing the field-independent part (i.e., carry-less multiplication) using a separate processor instruction.

Parts of our work have appeared at an earlier paper [25] and presentation [32]. References [25, 32] present the AES instruction proposal. This paper complements [25, 32] by discussing how our instructions can be implemented efficiently as well as how other algorithms like RSA and GCM can be accelerated within the context of TLS 1.2.

4. SSL/TLS, AES & RSA

To further understand our improvements, we first review the family of SSL/TLS protocols and the internals of the algorithms which we accelerate. Emphasis is placed on the 1.2 version of the TLS protocol because this version supports combined mode algorithms (i.e., Authenticated Encryption with Additional Data, AEAD) like GCM, which we study in this paper.

4.1 Anatomy of a Secure Sockets Layer Session

With the continuous growth of the World Wide Web [15], HTTPS has become the de facto standard for end-to-end secure communication where data sensitivity and privacy are required. The SSL/TLS protocol has evolved over the last 15 years to a state where it is the most widely deployed end-to-end security protocol today, with its primary use in protecting HTTP communications. The protocol can be divided into two parts - connection setup (also called handshake or control channel) and data exchange (also called data path or record protocol). Figure 1 illustrates the handshake, which is composed of different phases.

In phase 1, a client requests a secure connection from a server by sending a 'hello' message containing a unique random number (cookie, serving as unique identifier for this connection) and a list of supported cipher suites. If the server supports one of the offered cipher suites, it responds with its own cookie and the chosen cipher suite. The server also sends its certificate containing the server's identity and public key. The client verifies the server's certificate and extracts the server's public key. The client generates a random string, called a pre-master secret, and encrypts this using the server's public key (ensuring that only the server can decrypt this secret), sending this to the server in phase 3. The server decrypts the pre-master secret using its private key, which is a very compute-intensive operation. At this stage, the client and server share a secret value (pre-master secret) and a set of cookies, so are able to independently compute a set of cryptographic keys based on a well-defined formula called a Key Derivation Function (KDF).

In phases 5 and 6, the handshake ends with both parties sending 'authentication codes' based on the derived keying material, ensuring each side has computed the keys correctly. Now the data exchange phase can be started. Other functions of the TLS (e.g., session resumption) are not covered here.

Data is transferred within TLS using a record protocol. The record protocol breaks a data stream into a series of fragments, each independently protected. Before a fragment is transmitted, TLS protections are applied to each fragment. These include data authenticity, data confidentiality and optionally compression. A record header is added to the payload before sending down the network stack.

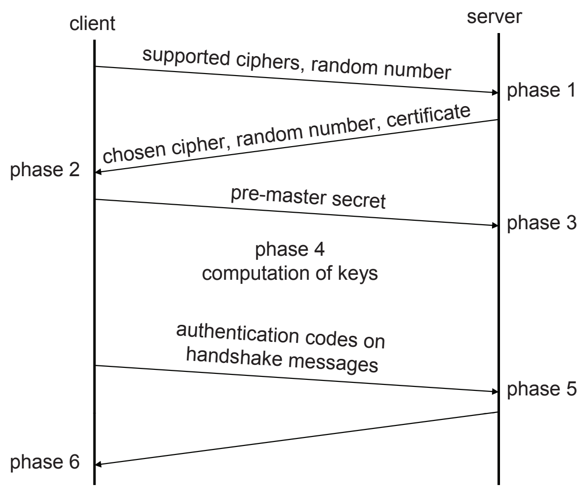

Figure 1: Secure Sockets Layer (SSL) Handshake

4.2 Combined Mode Support within TLS 1.2

TLS 1.2 introduces some subtle changes in the data path, where in the case of AEAD an explicit Initialization Vector (IV) is transmitted as part of a packet. TLS 1.2 supports AEAD based combined mode algorithms such as AES-128 GCM. Advantages of AEAD algorithms (e.g., AES GCM) over discrete mode algorithms (e.g., AES with HMAC-SHA1) is the ability to compute the cipher text and authentication tag using a single pass over the payload. Furthermore, using AES in the counter (CTR) mode allows efficient pipelining and parallelization, permitting concurrent crypto operations on multiple data blocks. Today, there are only few implementations supporting TLS 1.2 (GnuTLS & yaSSL claim supporting TLS 1.2, but they have not taken advantage of AEAD algorithms). Due to this limited deployment, the industry is unable to take advantage of the performance benefits offered by TLS 1.2. In this paper, we describe integration of TLS 1.2 and AES-128 GCM within the open source implementation of the OpenSSL library, in order to illustrate the resultant performance gains. Below, we elaborate on the subtleties of crypto algorithms and how they are optimized further from a networking performance and cost perspective.

4.3 AES

AES is the United States Government's standard for symmetric encryption, defined by FIPS 197 [1]. It is used in a large variety of applications where high throughput and security are required. In HTTPS, it can be used to provide confidentiality for the information that is transmitted over the Internet. AES is a symmetric encryption algorithm, which means that the same key is used for converting a plaintext to ciphertext, and vice versa. The structure of AES is shown in Figure 2.

AES first expands a key (that can be 128, 192, or 256 bits long) into a key schedule. A key schedule is a sequence of 128-bit words, called round keys, that are used during the encryption process. The encryption process itself is a succession of a set of mathematical transformations called AES rounds. During an AES round the input to the round is first XOR'd with a round key from the key schedule. The

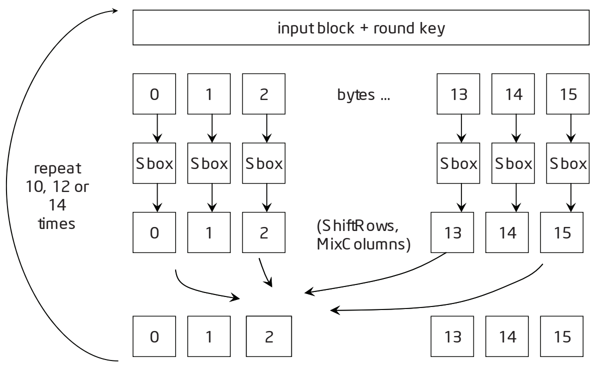

Figure 2: Structure of AES

exclusive OR (XOR) logical operation can also be seen as addition without generating carries.

In the next step of a round, each of the 16 bytes of the AES state is replaced by another value by using a non-linear transformation called SBox. The AES SBox consists of two stages. The first stage is an inversion, not in regular integer arithmetic, but in a finite field arithmetic based on the set $GF(2^8)$. The second stage is a bit-linear affine transformation (i.e., it can be implemented only with XOR gates). During encryption, the input x, which is considered an element of $GF(2^8)$; that is, an 8-bit vector, is first inverted, and then an affine map is applied to the result. During decryption, the input y=SBox(x) goes through the inverse affine map and is then inverted in $GF(2^8)$. The inversions just mentioned are performed in the finite field $GF(2^8)$, defined by the irreducible polynomial $p(x) = x^8 + x^4 + x^3 + x + 1$ or $0x11B$.

Next, the replaced byte values undergo two linear transformations called ShiftRows and MixColumns. The ShiftRows transformation is just a byte permutation. The MixColumns transformation operates on a matrix representation of the AES state. Each column is replaced by another one that results from a matrix multiplication. The transformation used for encryption is shown in Equation (1). In this equation, matrix-times-vector multiplications are performed according to the rules of the arithmetic of $GF(2^8)$ with the same irreducible polynomial that is used in the AES S-box, namely, $p(x) = x^8 + x^4 + x^3 + x + 1$.

$$\text{output} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \text{input} \tag{1}$$

During decryption, inverse ShiftRows is followed by inverse MixColumns. The inverse MixColumns transformation is shown in Equation (2)

$$\text{output} = \begin{bmatrix} 0xE & 0xB & 0xD & 0x9 \\ 0x9 & 0xE & 0xB & 0xD \\ 0xD & 0x9 & 0xE & 0xB \\ 0xB & 0xD & 0x9 & 0xE \end{bmatrix} \cdot \text{input} \tag{2}$$

The same process of the AES round is repeated 10, 12, or 14 times depending on the key size (128 , 192, or 256 bits). The last AES round omits the MixColumns transformation.

4.4 RSA

RSA is a public key cryptographic scheme. The main idea behind public key cryptography is that encryption techniques can be associated with secrets. Secrets are known only to at least one of the communicating parties and can simplify the decryption process. In public key cryptography, a message is encrypted using a public key. A public key is associated with a secret called the private key. Without knowledge of the private key, it is difficult to decrypt a message.

We further explain how public key cryptography works by presenting the RSA algorithm as an example. In this algorithm, the communicating parties choose two random large prime numbers p and q. For maximum security, p and q are of equal length. The communicating parties then compute the product:

$$n = p \cdot q.$$

Then the parties choose the public key e, such that the numbers e and $(p-1)\cdot(q-1)$ are relatively prime. The private key associated with the public key is a number d, such that:

$$e \cdot d \bmod (p-1)\cdot(q-1) = 1.$$

The encryption formula is simply:

$$C = M^e \bmod n,$$

where M is the plaintext and C is the ciphertext. The decryption formula is similarly:

$$M = C^d \bmod n.$$

The exponents d and e can be used interchangeably, meaning that encryption can be done by using d, and decryption can be done by using e.

RSA is typically implemented using the Chinese Remainder Theorem that reduces a single modular exponentiation operation into two operations of half length. Each modular exponentiation in turn is implemented using a square-and-multiply technique that reduces the exponentiation operation into a sequence of modular squaring and modular multiplication operations. Square-and-multiply may also be augmented with some windowing method for reducing the number of modular multiplications. Finally, modular squaring and multiplication operations can be reduced to big number multiplications by using reduction techniques such as Montgomery's or Barrett's [36, 16].

5. THE ACCELERATION TECHNOLOGIES

5.1 AES Acceleration

The AES round transformation is usually implemented using table lookups. Many software AES implementations [8, 3] use 8-16 tables of size 1K bytes. Four tables are typically used for encryption and four for decryption. Each table implements the SBox substitution transformation together with Galois Field multiplication operations. To complete an AES round, software implementations of AES perform several table lookups, each for a different byte of the cipher state, and XOR the results. This procedure results in a total of 16 table lookups. Its software cost is 24 clocks or 15 cycles per byte on a 3 GHz Intel® Core™ i7 processor, assuming all tables are in the first level cache. Other bit slice techniques for implementing AES reduce this cost but not substantially (e.g., 14 cycles per byte as reported in [40]).

As is evident from the numbers above, table lookup implementations of AES are not fast enough to saturate the high speed links (e.g., 10 Gbps) found in the Internet today. For example, for a 3 GHz processor core the AES cost of 15 cycles per byte translates to an AES processing throughput of 1.8 Gbps.

Table 1: AES Acceleration Instructions

Instruction	Description
AESENC	performs one round of an AES encryption flow operating on a 128-bit state and a 128-bit round key
AESENCLAST	performs the last round of an AES encryption flow operating on a 128-bit state and a 128-bit round key
AESDEC	performs one round of an AES decryption flow using the equivalent inverse cipher operating on a 128-bit state and a 128-bit round key
AESDECLAST	performs the last round of an AES decryption flow using the equivalent inverse cipher operating on a 128-bit state and a 128-bit round key

In this paper we introduce an alternative paradigm where block cipher rounds are implemented in combinatorial logic as part of the ALU data path of general purpose processor architecture [6]. Moreover this logic is exposed to the programmer as a set of instruction extensions. Using combinatorial logic as opposed to table lookups is a more efficient approach since time consuming memory lookup operations are avoided. Moreover the number of new instructions can be small and thus implementable and easy to validate. For example, for AES one needs only four new instructions. An example set of processor instructions that speedup AES is given in Table 1. These instructions are named after their functionality. These instructions are AESENC (AES round encryption), AESENCLAST (AES last round encryption), AESDEC (AES round decryption) and AESDECLAST (AES last round decryption).

The AESENC instruction implements the following transformations of the AES specification in the order presented: ShiftRows, SBox, MixColumns and AddRoundKey. The AESENCLAST implements ShiftRows, SBox and AddRoundKey but not MixColumns, since the last round omits this transformation. The AESDEC instruction implements inverse ShiftRows, inverse SBox, inverse MixColumns and AddRoundKey. Finally the AESDECLAST instruction implements inverse ShiftRows, inverse SBox, and AddRoundKey omitting the inverse MixColumns transformation.

The design of these new processor instructions is motivated by the structure of AES. This approach is different from off-loading the AES processing to a separate cryptographic processor. The AES instructions of Table 1 can be seen as cryptographic primitives for implementing not only AES but a wide range of cryptographic algorithms. This is because combinations of instruction invocations can be used for creating more generic mathematical primitives for performing computations in Galois Fields. For example, the combination of AESDECLAST and AESENC isolates the MixColumns transformation whereas the combination of AESENCLAST and AESDEC isolates the inverse MixColumns transformation. Using MixColumns and inverse MixColumns one can implement $GF(2^8)$ multiplication with any byte coefficient. One could argue that AES acceleration can also be provided with instructions that perform generic computations in Galois Fields. Such instructions, however would not achieve as good performance as our AES round instructions, due to the need for constructing the AES round results from the primitives.

Intel has implemented the instructions of Table 1 as extensions to the latest Intel® Core^{TM} micro-architecture [6]. The implementation of the AES instructions is pipelined in hardware. Hardware pipelining benefits modes of AES which are parallelizable such as the counter mode (CTR), the electronic codebook mode (ECB) and the decrypt mode of cipher block chaining (CBC). The only popular mode which cannot be parallelized is the encrypt mode of CBC. Hardware pipelining is used by the programmer to hide the latency of invoking the AES instructions by encrypting and decrypting multiple data blocks in parallel.

5.2 Using Composite Fields

The main implementation challenge associated with realizing the processor instructions of Table 1 is how to reduce the area requirement of the AES round logic so that it can fit into a general purpose processor architecture. General purpose processors are often associated with small area budgets for new features, due to the variety and complexity of the circuits they include (e.g., out-of-order execution pipelines, floating point processing logic, SIMD processing logic, etc.).

Clearly, the hardware implementation of the ShiftRows, MixColumns and affine map transformations is straightforward (i.e., wiring and/or a tree of XOR gates for each input bit). However, the complexity of computing the multiplicative inverse in $GF(2^8)$ which is part of SBox can be significant and is associated with the finite field arithmetic. In straightforward implementations of the AES SBox, the multiplicative inverse function and the affine map are combined in a single stage usually implemented via truth table logic, lookup memory or Binary Decision Diagram (BDD). While such approaches are potentially easier to conceptualize, they carry substantial gate-count cost required per SBox. Consider for example the straightforward implementation of a truth table as a sum of minterms. For an 8-bit input there are at most 256 minterms. On average there are 128. Taking the logical OR of 128 values requires 127 OR gates. Each of the minterms results from the logical AND of 8 inputs. Hence, the worst case number of gates required is $8 \cdot 8 \cdot 127 = 8,128$. More optimized implementations are reported in [38] requiring 2,623 2-input NAND gates for a table lookup logic and 2,818 2-input NAND gates for a BDD logic.

In this paper we argue that much of the SBox area requirement can be reduced using composite field technologies [41, 42, 38, 35]. A composite field is a finite field whose elements are vectors with coordinates in other smaller finite fields. An element of a field $GF(p^k)$, where $k = m \cdot n$, can be converted to the composite field $GF((p^m)^n)$ through an isomorphism. The main reason why composite field technologies reduce the area requirement of AES is because they associate a finite field element with its inverse through an easy-to-solve Cramer system. This system's solution requires a much smaller number of gates to implement.

```
Let $C = [c_{W-1} : c_{W-2} : ... : c_0]$ // sequence of words
Let $w$ be the word size
$\hat{N} \leftarrow -N^{-1} \bmod 2^w$
$U \leftarrow C$
for $i \leftarrow 0$ to $W/2-1$ do
    $u_i \leftarrow c_i \cdot \hat{N} \bmod 2^w,\ U \leftarrow U + u_i \cdot N \cdot 2^{i \cdot w}$
$U \leftarrow U/2^k$, if $U \geq N$ then $U \leftarrow U - N$
```

Figure 3: Montgomery Reduction

In what follows we justify our claim through an example. For the $GF((2^2)^4))$ composite field, the elements of the ground field $GF(2^2)$ are bit-pairs. Thus, its respective operations can be performed using only a handful (1-7) of logic gates. From the equation $a \cdot b = 1$ one can construct a Cramer system relating the bit pairs of the input with the bit pairs of the output. The resulting logic functions are simple and can be implemented with a few logic gates involving no more than 9 terms. In this example, the irreducible polynomial which extends the field $GF(2^2)$ to $GF((2^2)^4))$ is $x^4 + x^3 + x^2 + 2$. For this design the SBox area is no more than 419 gates. This corresponds to 84% gate count reduction as compared to the table lookup design of 2,623 2-input NAND gates. More area efficient SBox designs are reported in [18, 35]. If each SBox requires 419 gates, 16 SBoxes require 6704 gates. This means that the AES area requirement is in the same order of magnitude as that of other common ALU circuits.

5.3 RSA Acceleration

Another computationally expensive part of SSL transactions is RSA processing. The RSA algorithm involves the calculation of modular exponents for both the encryption and decryption processes. The calculation of modular exponents can be further reduced to performing modular multiplications and modular squaring operations using square-and-multiply or exponent windowing techniques.

A popular algorithm used for performing modular multiplications and modular squaring operations is the Montgomery algorithm. The Montgomery algorithm accepts as input two numbers A and B each of length k in bits and a divisor N and returns the number $C = A \cdot B \cdot 2^{-k} \bmod N$. Several ways to implement Montgomery have been proposed [30]. In our implementation the operands A and B are first multiplied with each other resulting in an intermediate value C. Then C is reduced mod$N$. For the reduction part the processing can be done on a word-by-word basis where the word size can vary from implementation to implementation. Assuming that $2k = w \cdot W$ where w is the word size, the Montgomery reduction can be written as in Figure 3.

The rationale behind the Montgomery reduction algorithm is that variable U is initialized to C and in every step of the iteration the least significant non-zero word of U becomes zero. In the end the most significant half of U is the desired result. Clearly, word-by-word Montgomery does not equal large number multiplication. However, if the word size w is half of the size of C (i.e., $W = 2$) Montgomery is reduced to 1.5 big number multiplications and one addition as shown in Figure 4.

```
$\hat{N} \leftarrow -N^{-1} \bmod 2^k$
$u \leftarrow (C \bmod 2^k) \cdot \hat{N} \bmod 2^k,\ C \leftarrow C + u \cdot N$
$C \leftarrow C/2^k$
if $C \geq N$ then $C \leftarrow C - N$ return $C$
```

Figure 4: Adapted Montgomery Reduction

From the above it is evident that the performance of RSA can be improved by accelerating the big number multiplication process which is an essential and compute-intensive part of the algorithm. Our implementation uses an optimized schoolbook big number multiplication algorithm. We have developed integer arithmetic software that can accelerate big number multiplication and modular reduction by at least 2x as compared to routines found in the crypto library of OpenSSL 0.9.8. Our software can be used not only in RSA public key encryption but also in Diffie Hellman key exchange and Elliptic Curve Cryptography.

The code listing of Figure 5 (assembly written using the AT&T syntax running on a Intel® Core^{TM} i7 processor) illustrates the main idea which is to do multiply and add operations combined with a register recycling technique for intermediate values. Here 'a' and 'b' are variables that hold the two large numbers to be multiplied (i.e., A and B) and the result C is stored in the variable 'r'. Partial products are computed in 'vertical order'. Vertical order means that all partial products between big number slices are computed together for the slices associated with the same index sum. Computations begin for the smallest index sum (i.e., 0) and continue all the way up to the largest (i.e., $2k - 2$). Each partial product is added to the final result as soon as it is computed. It is easy to show that carry propagation does not exceed the boundary of three big number slices. Hence for each partial product the code needs to invoke one 'mul' one 'add' and two add-with-carry 'adc' instructions. Register recycling helps in this case with reducing the 'mov' operations between registers and the system memory. Similarly we are able to accelerate other popular cryptographic schemes like RSA 2048 and elliptic curve cryptography. We have also investigated other techniques for big number multiplication, including Karatsuba-like constructions [46, 31] and found this schoolbook algorithm implementation to be the fastest.

5.4 GCM Acceleration

Another cryptographic component of protocols like SSL is message authentication. Message authentication can be supported by algorithms like HMAC-SHA1 but also modes of AES that combine encryption with authentication. One such mode is AES-GCM. In this section we argue that GCM can be sped up substantially with another processor instruction that performs carry-less multiplication [6].

Carry-less multiplication, also known as Galois Field Multiplication, is the operation of multiplying two numbers without generating or propagating carries. In the standard integer multiplication the first operand is shifted as many times as the positions of bits equal to '1' in the second operand. The product of the two operands is derived by adding the shifted versions of the first operand with each other. In carry-less multiplication the same procedure is followed except that additions do not generate or propagate carry. In

```
asm("mulq %3;\n"
    :"=a"(t0), "=d"(t1)
    :"a"(a[0]), "g"(b[0])
    :"cc");
  t2 = t0;
  t3 = t1;
  r[0] = t2;
  t2 = t3;
  t3 = t4;
  t4 = 0;
  asm("movq (%5), %%rax;\n\t"
      "mulq 8(%6);\n\t"
      "addq %3, %0;\n\t"
      "adcq %4, %1;\n\t"
      "adcq $0, %2;\n\t"
      "movq 8(%5), %%rax;\n\t"
      "mulq (%6);\n\t"
      "addq %3, %0;\n\t"
      "adcq %4, %1;\n\t"
      "adcq $0, %2;\n"
      :"+r"(t2), "+r"(t3), "+r"(t4), "=a"(t0), "=d"(t1)
      :"r"(a), "g"(b)
      :"cc");
r[1] = t2;
 asm("mulq %3;\n"
     :"=a"(t0), "=d"(t1)
     :"a"(a[1]), "g"(b[1])
     :"cc");
 asm("addq %2, %0;\n\t"
     "adcq %3, %1;\n"
     :"+r"(t0), "+r"(t1)
     :"r"(t3), "r"(t4)
     :"cc");
 r[2] = t0;
 r[3] = t1;
```

Figure 5: Big Number Multiplication Code

this way, bit additions are equivalent to the exclusive OR (XOR) logical operation.

We have implemented a fifth instruction supporting carry-less multiplication, named 'PCLMULQDQ'. Carry-less multiplication is supported between 64-bit quantities. This instruction demonstrates a latency of 14 clocks and a throughput of 10 clocks. In contrast, one of the fastest software techniques that perform the same operation known to us demonstrates a latency of approximately 100 clocks [8].

In what follows we justify our claim why we believe the PCLMULQDQ accelerates GCM. The most compute intensive part of GCM is multiplication in the finite field $GF(2^{128})$. The technique we describe in this paper is carried out in two steps: carry-less multiplication and reduction modulo $g = x^{128} + x^7 + x^2 + x + 1$. Carry-less multiplication can be performed through successive invocations of the PCLMULQDQ instruction.

To reduce a 256-bit carry-less product modulo a polynomial g of degree 128, we first split it into two 128-bit halves. The least significant half is simply XOR-ed with the final remainder (since the degree of g is 128). For the most significant part, we develop an algorithm that realizes division via two multiplications. This algorithm can be seen as an extension of the Barrett reduction algorithm [16] to modulo-2 arithmetic, or as an extension of the Feldmeier CRC generation algorithm [22] to dividends and divisors of arbitrary size.

Since we do not take into account the least significant half of the input (see above), we investigate the efficient generation of a remainder $p(x)$ defined as follows:

$$p(x) = c(x) \cdot x^t \bmod g(x)$$

where, $c(x)$ is a polynomial of degree $s-1$ with coefficients in $GF(2)$, representing the most significant bits of the carry-less product (for GCM, $s = 128$), t is the degree of the polynomial g (for GCM, $t = 128$), and $g(x)$ is the irreducible polynomial defining the final field (for GCM, $g = g(x) = x^{128} + x^7 + x^2 + x + 1$).

Our algorithm involves the following steps:

Preprocessing: For the given irreducible polynomial g two polynomials g^* and q^+ are computed first. The polynomial g^* is of degree $t-1$ consisting of the t least significant terms of g, whereas the polynomial q^+ is of degree s and is equal to the quotient of the division of x^{t+s} with the polynomial g.

Calculation of the remainder polynomial: Step 1: The input c is multiplied with q^+. The result is a polynomial of degree $2s-1$. Step 2: The s most significant terms of the polynomial resulting from step 1 are multiplied with g^*. The result is a polynomial of degree $t+s-2$. Step 3: The algorithm returns the t least significant terms of the polynomial resulting from step 2. This is the desired remainder.

One can see that the quotient from the division of x^{256} with g is g itself. The polynomial $g = g(x) = x^{128} + x^7 + x^2 + x + 1$ contains only 5 non-zero coefficients (therefore also called 'pentanomial'). This polynomial can be represented as [1 :< 120 zeros >: 10000111]. Multiplying this carry-less with a 128 bit value and keeping the 128 most significant bits can be obtained by: (i) Shifting the 64 most significant bits of the input by 63, 62 and 57 bit positions to the right. (ii) XOR-ing these shifted copies with the 64 least significant bits of the input. Next, we carry-less multiply this 128-bit result with g, and keep the 128 least significant bits. This can be done by: (i) shifting the 128-bit input by 1, 2 and 7 positions to the left. (ii) XOR-ing the results. Hence we split the finite field multiplication into a field-independent part which we perform with PCLMULQDQ and a field dependent part which we reduce to a small number of shift and XOR operations. Using this instruction we achieve a GCM performance of approximately 2.6 cycles per byte or 3.9 cycles per byte together with AES as discussed later. This is about 3 times faster than typical HMAC-SHA-1 implementations.

6. A HIGH PERFORMANCE TLS PROTOCOL STACK

6.1 Implementing the 1.2 Version

As the OpenSSL library only provides TLS implementations up to version 1.0, we have added the necessary changes needed to have a functional TLS 1.2 version. The following are the major differences that need to be implemented between TLS 1.0 and TLS 1.2. First, key generation is different in TLS 1.2. Key generation entails the creation of keys for the record protocol of TLS from the security parameters provided by the handshake phase. The major difference between TLS 1.0 and TLS 1.2 is in the Pseudo Random Function (PRF) employed. Specifically, the MD5/SHA-1 combination in TLS 1.0 is replaced by SHA-256 or stronger hashes. In addition in TLS 1.2, the PRF is negotiable during the control channel handshake.

Second, there is difference in the IV generation between TLS 1.2 and TLS 1.0 or prior versions. In the context of AEAD algorithms (e.g., AES-GCM) which are introduced in TLS 1.2 IV consists of two portions: an implicit part coming from the key generation and an explicit part that needs to be unique. This may not be the case for other ciphers or for versions prior to TLS 1.2. Other changes introduced in TLS 1.2 include error handling, certificate handling and deprecation of older cipher suites. Perhaps the most important difference between TLS 1.0 and TLS 1.2 from a performance perspective is the introduction of algorithmic agility which allows us to benefit from AEAD algorithms (e.g., AES-GCM). We focus on this class of algorithms in the following subsection.

6.2 Use of AEAD Algorithms in the TLS stack

In versions of TLS prior to 1.2, discrete mode algorithms are employed which provide confidentiality and message authentication. Each algorithm uses an independent key. AEAD achieves both encryption and authentication using a single key.

There is significant difference regarding the integration of AEAD and non-AEAD algorithms in the TLS 1.2 protocol. For AEAD ciphers an explicit IV value is generated and transmitted with each payload. Furthermore an implicit IV is created from the key generation process, concatenated with the explicit IV and used as input to the algorithm. In the data path, non-AEAD ciphers apply a message authentication code before encryption. Then they encrypt and transmit the data. The MAC is encrypted in this case. However, for AEAD, this operation is reversed. Encryption happens prior to MAC generation. The MAC is not encrypted in this case.

We implemented the Advanced Encryption Standard (AES) algorithm in the Galois/Counter Mode (GCM) with a 128-bit key. For the explicit IV we use the recommended method of [33] (3.2.1). In the context of AES, the IV is comprised of 8 bytes of explicit IV and 4 bytes implicit IV. The explicit IV is the sequence number that is sent and retrieved together with the payload, whereas the implicit IV is derived from the key generation process. By including a sequence number in the IV, we can satisfy the requirement that the IV values are unique.

AEAD allow additional data to be authenticated. In the case of TLS 1.2 this additional data comprises a sequence number, packet type, TLS version and the compressed packet length.

Figure 6 shows how the fields of the data packet map onto the inputs and outputs of the cipher AES-128 GCM. As indicated in the figure, the data field is encrypted and authenticated, and is carried along with a header and a sequence number. The header is authenticated by being included in the authenticated data. The sequence number is included in the IV. The authentication tag is carried along with the encrypted data in the packet payload. Our implementation is available in the public domain [14] for experimentation.

7. RESULTS

In what follows we describe the results from our experiments on measuring the performance of various cryptographic tasks and TLS banking worloads. We present results at an instruction level, function level and session level.

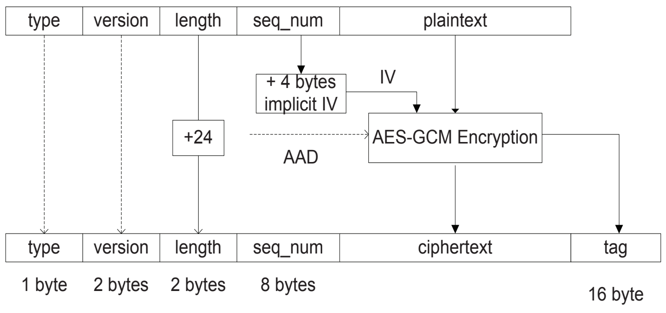

Figure 6: AES-128 GCM Packet Format

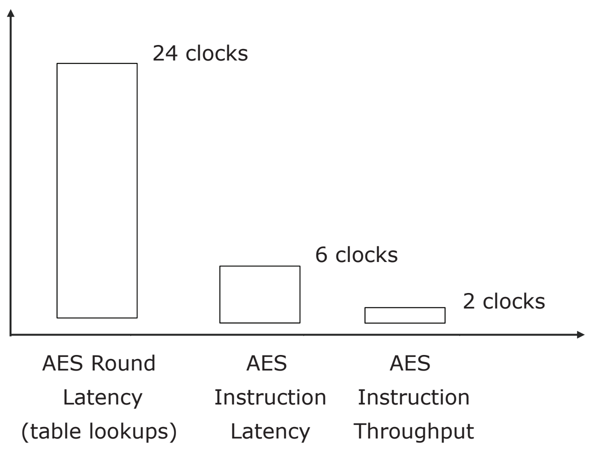

Figure 7: Instruction Level Performance

Instruction level performance pertains to the latency of invoking the AES round instructions. Function level pertains to the cost of various modes of AES and RSA encryption for a given input data set. Session level pertains to the performance measurements taken in the context of an overall TLS session. We dissect and describe the different cryptographic components that contribute to the overall cost of connecting and maintaining a TLS session. We also demonstrate that our technologies result in substantial performance improvement. We compare against optimized cryptographic implementations we developed ourselves, as well as OpenSSL. We used OpenSSL since it is hard to get access to other proprietary cryptographic algorithm implementations. So, we used one of the best suites available in the public domain.

7.1 AES Instruction Level

Figure 7 presents a performance comparison between an AES software round implementation that does not use our instructions vs. one that leverages the instructions introduced in Section 5. This data is gathered using a 3 GHz Xeon® processor with our instruction extensions.

The leftmost bar in the figure depicts the latency of an AES round using the table lookup method. The next bar depicts the latency of completing an AES round in combinatorial logic. The rightmost bar depicts the AES instruction throughput. By instruction 'throughput' we mean the minimum time elapsed between the completion of two independent AES round operations, which is smaller than the instruction latency since the AES circuit is pipelined. Specifically, for each separate data block requiring an AES operation, this is the time between the completion of an AES operation on one block and the completion of the same AES operation on another data block.

For the table lookup implementation, we observe a latency of 24 clocks per round. This requires table lookups for all bytes of the cipher state. The minimum time between the completion of independent round operations for this approach is approximately 24 clocks too. The software used to perform these measurements was an optimized AES implementation, based on the code of Brian Gladman.

When performing AES rounds using our instructions, we find that the round latency is reduced to 6 clocks, returning a 4x improvement. Furthermore, the throughput decreases to 2 clocks, returning a 12x improvement. These measurements relate to both AES encrypt and decrypt round operations as well as encrypt and decrypt last rounds.

7.2 AES Function Level

In our tests we compare four representative modes of operation of AES. Because of the difference in the way AES is used in each mode, performance varies across modes of operation. The first three modes (CBC, CTR, ECB) support encryption only. GCM supports encryption and message authentication. The cycles per byte presented here are in accordance with the instruction level results shown in Figure 7. Instruction latencies are applied to the full 10 round implementation of AES-128 and amortized over a 16-byte AES block.

Table 2: Algorithm Level Performance

algorithm mode	table lookups (cycles/byte)	AES instructions (cycles/byte)
AES-128 CBC encrypt	16.1	4.1
AES-128 CTR encrypt	19.3	1.3
AES-128 ECB encrypt	15.6	1.2
AES-128 GCM encrypt	29.5	3.9

AES in the CBC mode results in a performance of 16.1 cycles per byte using the table lookup method. This is reduced to 4.1 cycles per byte when enabling the AES instructions. This illustrates a 4x performance gain when leveraging our AES hardware over pure software without it. CBC is not a parallelizable mode since each input to a subsequent block operation requires the output of the previous block. Because of this CBC cannot take advantage of the small instruction throughput shown in Figure 7.

AES in CTR and ECB modes displays the best results with a 14x/13x performance improvement respectively. In these modes AES leverages the round instructions in the most optimal way. In CTR and ECB each block can be independently encrypted/decrypted without reliance on the previous or next block, hence allowing efficient hardware parallelization and pipelining. In the code used for these measurements we encrypt/decrypt four blocks at a time. The table lookup implementation of CTR demonstrates a cost of 19.3 cycles per byte. CTR is slower than CBC and ECB because of the need to perform an additional byte shuffling operation for endianness compliance for every block. The performance of ECB is 15.5 cycles per byte. When using our instructions the performance of CTR becomes 1.3 cycles per byte, whereas the performance of ECB becomes 1.2 cycles per byte. Due to hardware pipelining the effective latency per round is only 2 clocks for these modes. In addition due to the fact that our instruction implementation is done in the SIMD domain of the Intel® Core^{TM} micro-architecture, the byte shuffling required by the CTR mode is done efficiently using the PSHUFB instruction.

The results for AES-GCM are equally good, illustrating a 7.5x gain over the table lookup implementation. GCM removes the need for a separate data authenticity function within the context of a protocol such as TLS. This improvement becomes critical when comparing our GCM code (i.e, combined encryption and authentication) against the 'traditional' discrete model of using AES-CBC and HMAC-SHA1. SHA-1 costs approximately 8 cycles per byte. GCM processing with the PCLMULQDQ instruction can be accomplished at an additional cost of only 2.6 cycles per byte. The table lookup approach consumes 10 cycles per byte for GCM [5]. The increase from 1.3 to 3.9 cycles per byte for this algorithm is due to the additional Galois Field multiplication operation required. For an optimized software implementation of AES-GCM, without the instructions, the typical cost is 29.5 cycles/byte. By comparison, our acceleration technologies reduce this to 3.9 cycles/byte. Saturation of a 10 Gbps link without our technologies requires 12 3.0 Ghz cores. Our optimizations bring this down to just over a single core, leaving the remaining cores for other critical workloads. This analysis excludes the RSA overhead for key establishment. Later in the paper we take this overhead into account.

Acceleration gains are similar for other key sizes. For AES-192 and AES-256, the CBC speedup is 4x and the GCM speedup is 7.5x. AES-192 operates at 4.9 cycles/byte for CBC and 4.2 cycles/byte for GCM. AES-256 operates at 5.6 cycles/byte for CBC and 4.5 cycles/byte for GCM.

7.3 RSA Function Level

The RSA performance depends on the performance of the underlying integer multiplication building blocks. In what follows we compare the performance of routines that perform 512 by 512, 1024 by 1024, and 2048 by 2048 bit multiplication coming from the OpenSSL libraries (0.9.8 and 1.0 versions) and our code. We have implemented two routines: One based on the single iteration Karatsuba multiplication variant described in [31] and one based on our optimized schoolbook technique. Our results are shown in Table 3. As is evident from the table, our schoolbook multiplication routines outperform both the Karatsuba code and the multiplication code (schoolbook) from the OpenSSL library. For example, the 512 by 512 bit multiplication can be completed in only 257 clocks, whereas Karatsuba needs 434 clocks and OpenSSL schoolbook 611 clocks.

Next we compare the RSA 1024 and RSA 2048 performance at a private decrypt operation level coming from OpenSSL 0.9.8, 1.0, our code using Karatsuba multiplication and our code using our optimized schoolbook technique. Our results are shown in Table 4. The numbers mean private decrypt operations per second per 3 GHz Intel® Core^{TM} i7 processor core.

The main difference between OpenSSL 0.9.8 and OpenSSL 1.0 is that the 1.0 version implements the word-by-word

Table 3: Performance of Integer Multiplication (processor clocks)

	512 by 512 bit	**1024 by 1024 bit**	**2048 by 2048 bit**
OpenSSL 0.9.8, 1.0	611	1937	6212
our code (Karatsuba)	434	1309	5024
our code (schoolbook)	257	1052	3815

Table 4: Performance Comparison (private decrypt operations per second)

	RSA 1024 (private decrypt)	**RSA 2048 (private decrypt)**
OpenSSL 0.9.8	1463	259
OpenSSL 1.0	2143	360
our code (Karatsuba)	1893	375
our code (schoolbook)	2990	454

Montgomery reduction using optimized assembly code that reduces the number of 'mov' operations between registers and memory. As is evident from the tables the best RSA performance comes from our code when using our schoolbook multiplication technique. This results in a performance of 2990 RSA 1024 private decrypt operations per second per 3 GHz processor core. The main departure from the OpenSSL techniques is that in our code Montgomery is done in a single step as opposed doing the reduction on a word-by-word basis. In this way, Montgomery is reduced to 1.5 big number multiplications, where the multiplications themselves are performed using our optimized schoolbook code evaluated in Table 3.

The acceleration gain coming from our code is also substantial in other processors as well. In a Pentium® 4 architecture our code speeds up RSA 1024 from 881 private decrypt operations per second per 3 GHz core to 981 operations per second per core. Multiplication between 512-bit integers takes 806 clocks. The Pentium® 4 architecture is slower because the 64-bit mode is emulated.

7.4 Session Level

We use two methods to measure the TLS session level performance.

First, we utilize Oprofile [9] and a microbenchmark tool to obtain a cost breakdown of a typical TLS banking session of file size equal to 140KB. Our microbenchmark tool uses an optimal table lookup AES implementation. The cost breakdown result is illustrated in Figure 8.

As can be seen from the leftmost bar of this diagram, the cost of the RSA 1024 asymmetric key operation for a given session (2.17 million clocks) is almost equal to the cost of the AES-128 CBC operation (2.30 million clocks) for a 140KB data file. This in turn is roughly equal to the cost of the HMAC-SHA1 operation and other overheads such as OS, and networking overheads (1.18+0.73 million clocks). By

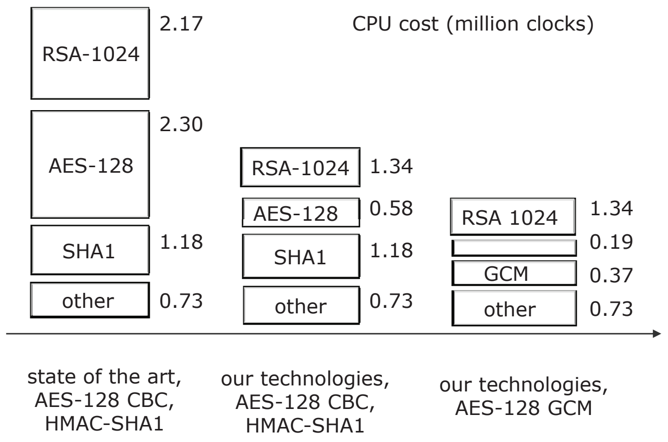

Figure 8: Session Level Performance

accelerating the symmetric key operation, as well as optimizing the asymmetric key operation, we are impacting nearly 90% (5.65 over 6.38 million clocks) of the crypto overhead for the overall TLS session cost.

In the next bar of this diagram, we illustrate the benefit coming from optimizing the RSA operation and also leveraging the AES instructions for encryption and decryption. This alone provides a 1.67x improvement over the number of TLS sessions per second that can be achieved over a non-optimal TLS implementation. AES-128 is still in the CBC mode for this bar.

In the rightmost bar of the diagram, we replace the cipher suite of AES-128 CBC and HMAC-SHA1 with AES-128 GCM which is a combined mode that provides the best performance results, illustrating a 2.43X improvement over the leftmost bar.

The microbenchmark is built using our own code, including the state-of-the-art baseline. Now let's consider our optimizations in the context of a public domain open source TLS stack. Our second method is to utilize the OpenSSL SSL/TLS performance timing program called *s_time* in order to benchmark our TLS improvements. The *s_time* command implements a client which connects to a server using SSL/TLS. It requests a file from the server and measures the time to transfer the payload data. Networking latencies are not included in the measurements. It reports the number of connections within a given time frame, the amount of data transferred and the average user CPU time spent for each connection. Our results are presented in Table 5.

We see approximately 1200 sequential connections per second of user time for the cipher suite RSA-1024, AES-128 GCM as compared to 770 connections per second of user time for the cipher suite RSA-1204, AES-128 CBC, HMAC-SHA1 using our optimized RSA and AES implementations, and 200 connections per second of user time for the same cipher suite without our optimizations for the same file size of 140KB. The result is a performance gain of 6x allowing for many more connections to be supported on a given server. It is worth pointing out that the best absolute performance coming from our two measurement methodologies is the same (i.e., approximately 1200 transactions per second per core).

Table 5: TLS Stack Performance from *s_time*

cipher suite	connections/sec
AES-128 CBC, SHA1, state-of-the-art	199
AES-128 CBC, SHA1, our technologies	768
AES-128 GCM, our technologies	1199

7.5 Encrypting the Internet

So far we have described how improvements in the performance of cryptographic algorithms can speed up the performance of SSL servers using the TLS 1.2 protocol stack. To justify our claim for end-to-end Internet encryption, we answer a different question: How many processor cores in general purpose hardware do we need to saturate a high speed Internet link using our technologies? We examine 1 and 10 Gbps links.

The single core processing throughput (measured in Gbps) associated with a set of crypto technologies can be computed from the total number of transactions per second of Table 5 and the session size (in bits). Specifically:

$$\text{throughput} = \text{sessions per second} \times \text{session size}$$

From this throughput computation we can derive the number of 3 GHz processor cores needed for saturating 1 and 10 Gbps network links by dividing the link capacity with the processing throughput associated with each technology. Our results are shown in Table 6. The numbers assume good scaling of the SSL/TLS performance across multiple cores, which we believe is feasible given the inherent parallelization support of the protocol across different sessions.

Table 6: Cores to Saturate 1 (10) Gbps

cipher suite	processing throughput (Gbps)	cores to saturate 1 (10) Gbps
AES-128 CBC, SHA1, state of the art	0.22	5 (46)
AES-128 CBC, SHA1, our crypto technologies	0.86	2 (12)
AES-128, GCM, our crypto technologies	1.34	1 (8)

As is evident from the table it is now possible to take general purpose hardware, e.g., with two sockets and six cores per socket and our instruction extensions and provide equivalent functionality to a dedicated appliance allowing 9,600 ($= 8 \times 1200$) transactions per second at 10 Gbps, while still leaving four cores for application processing. The system uses 8 cores with encryption. If no encryption was applied, the system would use 3 cores. In both cases a substantial number of cores (4-9) is left for other tasks.

At the time of writing, dedicated appliances tend to be more expensive than general purpose systems. For example, appliances reported in reference [12] range from $18,850 to $36,990. Also at the time of writing, a general purpose blade server [10] can cost up to $3,999. Since the performance of appliances varies considerably, we use a normalized metric to express their cost efficiency as it pertains to SSL throughput. Specifically, we divide the cost of a device by its associated SSL transaction rate. For a hypothetical appliance that costs $18,850 and supports 4,500 banking transactions per second, the normalized cost is $4. On the other hand the normalized cost of the afore-mentioned blade server, supporting our technologies, is 28 cents only, for the same SSL transaction size. Such substantial cost reduction could allow network security technologies to be deployed in areas where previously it was considered cost inhibited.

8. CONCLUSIONS

In this paper we argued about the need for encrypting and authenticating Internet transactions. We analyzed the performance of some cryptographic algorithms and proposed techniques to speed them up significantly. Our solution is primarily based on adding small hardware extensions to general purpose architectures. We supported our claims with performance results demonstrating the efficiency of using general purpose hardware for doing SSL/TLS traffic processing at 10 Gbps line rates.

Our work is far from done however. Moving to 128-bits equivalent security for public key schemes may require further innovations for increasing the performance of public key encryption. Furthermore, NIST organizes a hash competition for defining the next SHA-3 standard. Understanding how to accelerate the winning algorithm will be important as well. Third, public trials will be useful for further validating our results.

9. REFERENCES

[1] "Advanced Encryption Standard". Website. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[2] "Cisco WebVPN Services Module - Cryptographic Accelerator". Website, hardware.com. http://us.hardware.com/store/cisco/WS-SVC-WEBVPN-K9=/campaign/1-85819001.

[3] "Crypto++". Crypto++ Website. http://www.cryptopp.com.

[4] "Data-stealing Malware on the Rise, Solutions to Keep Businesses and Consumers Safe". Website. http://us.trendmicro.com/imperia/md/content/us/pdf/threats/securitylibrary/data_stealing_malware_focus_report_-_june_2009.pdf.

[5] The Galois/Counter Mode of Operation (GCM). Website, NIST. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf.

[6] "Intel AVX, Intel Software Network". Intel Website. http://software.intel.com/en-us/avx/.

[7] "Internet Passes 600,000 SSL Sites". Website, SSL Shopper. http://www.sslshopper.com/article-internet-passes-600000-ssl-sites.html/.

[8] "OpenSSL Library". OpenSSL Website. http://www.openssl.org.

[9] "OProfile". OProfile Website. http://oprofile.sourceforge.net/news/.

[10] "PowerEdge Rack Servers". Website, dell.com. http://www.dell.com/us/en/gen/servers/rack_optimized/cp.aspx?refid=rack_optimized&s=gen.

[11] "SSL Acceleration and Offloading: What Are the Security Implications?". Website, WindowSecurity.com. http://www.windowsecurity.com/articles/SSL-Acceleration-Offloading-Security-Implications.html.

[12] "SSL Decryption and Re-encryption". Website,

zeus.com. http://www.zeus.com/products/traffic-manager/secure/ssl.html.

[13] "The Total Number of Web Sites on Earth". Website, Get Netted. http://www.wlug.net/the-total-number-of-websites-on-earth/.

[14] "TLS 1.2 Open Source Release". Website. http://www.mail-archive.com/openssl-dev@openssl.org/msg27172.html.

[15] "Two Year Study of Global Internet Traffic, NANOG47". Website, Internet Society. http://isoc-dc.org/wordpress/?p=920.

[16] P. Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor". *Masters Thesis, University of Oxford, UK*, 1986.

[17] A. Bosselaers, R. Govaerts, and J. Vandewalle. "Comparison of Three Modular Reduction Functions". *Proceedings, Advances in Cryptology (CRYPTO 1993)*, 1993.

[18] D. Canright. "A Very Compact S-Box for AES". *Proceedings, Workshop on Cryptographic Hardware and Embedded Systems (CHES 2005)*, 2005.

[19] A. J. Elbirt. "Fast and Efficient Implementation of AES via Instruction Set Extensions". *Proceedings, 21st International Conference on Advanced Information Networking and Applications Workshops*, 2007.

[20] N. Farrell. "google tightens Gmail security". Website, January 2010. http://www.theinquirer.net/inquirer/news/1586138/google-tightens-gmail-security.

[21] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen. "AES Implementation on a Grain of Sand". *IEE Proceedings on Information Security*, 2005.

[22] D. Feldmeier. "Fast Software Implementation of Error Detection Codes". *IEEE Transactions on Networking*, pages 640–651, 1995.

[23] A. M. Fiskiran and R. B. Lee. "On Chip Lookup Tables for Fast Symmetric Key Encryption". *Proceedings, IEEE International Conf. on Application-Specific Systems, Architectures and Processors*, pages 356–363, 2005.

[24] K. Grewal and M. Miller. "Next Generation Scalable, Cost-effective E2E Security". RSA Conference, 2010.

[25] S. Gueron. "Intel's New AES Instructions for Enhanced Performance and Security". *Proceedings, 16th International Workshop on Fast Software Encryption (FSE 2009), LNCS 5665*, pages 51 – 66, 2009.

[26] A. Hodjat, D. Hwang, B.-C. Lai, K. Tiri, and I. Verbauwhede. "A 3.84 Gbits/s AES Crypto Coprocessor with Modes of Operation in a 0.18-μm CMOS Technology". *Proceedings, 15th ACM Great Lakes Symposium on VLSI*, pages 60–63, 2005.

[27] A. Hodjat and I. Verbauwhede. "A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA". *Proceedings, 12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, pages 308–309, 2005.

[28] D. Knuth. "Seminumerical Algorithms". *The Art of Computer Programming, Addison-Wesley*, 2, 1997.

[29] C. K. Koc. "Analysis of Sliding Window Techniques for Exponentiation". *Computers and Mathematics with Application*, 30(10):17–24, 1995.

[30] C. K. Koc, T. Acar, and B. S. Kaliski. "Analyzing and Comparing Montgomery Multiplication Algorithms". *IEEE Micro*, 16(3):26–33, 1996.

[31] M. Kounavis. "A New Method for Fast Integer Multiplication and its Application to Cryptography". *Proceedings, 2007 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2007.

[32] M. Kounavis and L. Xu. "AES-NI: New Technology for Improving Encryption Efficiency and Enhancing Data Security in the Enterprise Cloud". Intel Developer Forum, 2009. https://intel.wingateweb.com/us09/scheduler/sessions.do?searchGroup=9&searchGroupID=10133&profileItem_id=10004.

[33] D. McGrew. "An Interface and Algorithms for Authenticated Encryption". Website, January 2008. http://www.faqs.org/rfcs/rfc5116.html.

[34] A. Menezes, P. Oorschot, and S. Vanstone. *"Handbook of Applied Cryptography"*. CRC Press, 1997.

[35] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. "A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box". *Proceedings of CT-RSA 2005*, 2005.

[36] P. Montgomery. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor". *Masters Thesis, University of Oxford, UK*, 1986.

[37] P. Montogomery. "Five, Six and Seven-term Karatsuba-like Formulae". *IEEE Transactions on Computers*, 2005.

[38] S. Moriokah and A. Satoh. "An Optimized S-Box Circuit Architecture for Low Power AES Design". *Proceedings, Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, pages 172–186, May 2002.

[39] K. K. Peretti. "Data Breaches: What the Underground World of Carding Reveals". *the Santa Clara Computer and High Technology Journal*, 25(2):375 – 413, January 2009.

[40] C. Rebeiro, D. Selvakumar, and A. S. L. Devi. "Bitslice Implementation of AES". *Cryptology and Network Security, LNCS 4301*, 2006.

[41] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi. "Efficient Rijndael Encryption with Composite Field Arithmetic". *Proceedings, Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)*, pages 175 – 188, May 2001.

[42] A. Satoh, S. Moriokah, K. Takano, and S. Munetoh. "A Compact Rijndael Hardware Architecture with SBox Optimization". *Lecture Notes in Computer Science, LNCS 2248*, pages 239–254, 2001.

[43] S. Schillace. "Default HTTPS Access for gmail". Website, January 2010. http://gmailblog.blogspot.com/2010/01/default-https-access-for-gmail.html.

[44] SecurityFocus. "Data Breach Costs Rise, Response Costs Fall". Website, February 2009. http://www.securityfocus.com/brief/900.

[45] I. Verbauwhede, P. Schaumont, and H. Kuo. "Design and Performance Testing of a 2.29 Gb/s Rijndael Processor". *IEEE Journal of Solid-State Circuits*, pages 569–572, 2003.

[46] A. Weimerskirch and C. Paar. "Generalizations of the Karatsuba Algorithm for Efficient Implementations. *Technical Report, University of Ruhr, Bochum, Germany*, 2003.

[47] A. Whitten. "HTTPS Security for Web Applications". Website, June 2009. http://googleonlinesecurity.blogspot.com/2009/06/https-security-for-web-applications.html.

[48] J. Wolkerstorfer, E. Oswald, and M. Lamberger. "An ASIC Implementation of the AES SBoxes". *Proceedings, CT-RSA 2002*, 2002.