ACCENT: Cognitive Cryptography Plugged Compression for SSL/TLS-Based Cloud Computing Services

KI-WOONG PARK and KYU HO PARK, Korea Advanced Institute of Science and Technology

Emerging cloud services, including mobile offices, Web-based storage services, and content delivery services, run diverse workloads under various device platforms, networks, and cloud service providers. They have been realized on top of SSL/TLS, which is the de facto protocol for end-to-end secure communication over the Internet. In an attempt to achieve a cognitive SSL/TLS with heterogeneous environments (device, network, and cloud) and workload awareness, we thoroughly analyze SSL/TLS-based data communication and identify three critical mismatches in a conventional SSL/TLS-based data transmission. The first mismatch is the performance of loosely coupled encryption-compression and communication routines that lead to underutilized computation and communication resources. The second mismatch is that the conventional SSL/TLS only provides a static compression mode, irrespective of the dynamically changing status of each SSL/TLS connection and the computing power gap between the cloud service provider and diverse device platforms. The third is the memory allocation overhead due to frequent compression switching in the SSL/TLS. As a remedy to these rudimentary operations, we present a system called an Adaptive Cryptography Plugged Compression Network (ACCENT) for SSL/TLS-based cloud services. It is comprised of the following three novel mechanisms, each of which aims to provide an optimal SSL/TLS communication and maximize the network transfer performance of an SSL/TLS protocol stack: tightly-coupled threaded SSL/TLS coding, floating scale-based adaptive compression negotiation, and unified memory allocation for seamless compression switching. We implemented and tested the mechanisms in OpenSSL-1.0.0. ACCENT is integrated into the Web-interface layer and SSL/TLS-based secure storage service within a real cloud computing service, called *iCubeCloud*, as the key primitive for SSL/TLS-based data delivery over the Internet.

Categories and Subject Descriptors: C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*Internet*

General Terms: Design, Performance

Additional Key Words and Phrases: Communication system security, compression, cryptography, SSL/TLS

ACM Reference Format:
Park, K.-W. and Park, K. H. 2011. ACCENT: Cognitive cryptography plugged compression for SSL/TLS-based cloud computing services. ACM Trans. Internet Technol. 11, 2, Article 7 (December 2011), 31 pages.
DOI = 10.1145/2049656.2049659 http://doi.acm.org/10.1145/2049656.2049659

1. INTRODUCTION

Cloud computing is an important transition and a paradigm shift in Internet technology. In recent years, emerging cloud services, including mobile offices, Web-based storage services, and content delivery services have become popular. Examples include GoogleDocs [GoogleDocs 2011], Dropbox [Dropbox 2011], Amazon EC2 [Amazon-EC2

The work was supported by the Ministry of Knowledge Economy, Republic of Korea (MKE), Project No. A1100-0901-1710.
Authors' address: K.-W. Park and K. H. Park, Department of Electrical Engineering, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea; email: K.-W. Park, woongbak@core.kaist.ac.kr; K. H. Park, kpark@ee.kaist.ac.kr.

DOI 10.1145/2049656.2049659 http://doi.acm.org/10.1145/2049656.2049659

2011], S3 [Amazon-S3 2011] and Microsoft Azure [Microsoft-Azure 2010]. They are generally equipped with SSL/TLS for data confidentiality and integrity. Although these security systems furnish various types of Internet services, they have a fundamental common purpose: to transfer data with end-to-end confidentiality and integrity via SSL/TLS and network subsystems where a wide spectrum of computing devices (smartphones, tablets, netbooks, and desktop PCs) and networks are connected to each other [NetCraft 2010]. Given that the cloud-based services often require users to be connected to access their information, the end-user experience of the service can be affected by the dynamics of the network and the computation workload [John Dunlop and Abani 2010].

From the perspective of communication efficiency, data compression is an effective way of enabling high-performance data delivery over SSL/TLS. However, only about 4.6% of all SSL/TLS Web sites are compression-enabled [W3Techs 2011], mainly because the selection of a certain compression method may degrade performance and user experience where the network and computation workload are diverse and dynamic. In addition, to fully understand and optimize the impact of compression, we need to consider the associated trade-off between computation and communication since compression techniques vary in terms of performance, particularly with regard to the compression ratio, compression time, and decompression time [Gilchrist 2009]. Otherwise an SSL/TLS-based system may encounter a computation or network bottleneck. That is, there is no single SSL/TLS encoding scheme that fits all due to environmental heterogeneity (device, network, and cloud) and workload. This type of problem makes it more challenging to cognitively apply compression to SSL/TLS connections. As a result, we need novel mechanisms that can automatically and transparently identify the most profitable SSL/TLS encoding technique.

In order to improve the performance of SSL/TLS-based systems, most studies have focused on optimizing an individual network and computation utilization [Wu et al. 2001; Alaidaros et al. 2007; Morales-Sandoval and Feregrino-Uribe 2005; Kounavis et al. 2010; Okamoto et al. 2003; Shacham et al. 2004] or improving system performance from the perspective of the server rather than the end-to-end perspective from the client to the server [Coarfa et al. 2006; Chou 2002; Castelluccia et al. 2006; Jang et al. 2011]. Comparatively few studies have applied a global view to the investigation of performance improvement, network and computation utilization, and deliberation of end-to-end network and system status.

The following issues, for instance, have been ignored in the literature: how SSL/TLS computation and communication routines affect network performance; the most effective reaction of SSL/TLS if a network or computation workload is changed or if there is a wide gap in computing power between the sender and receiver; and how the best compression method is identified transparently in an SSL/TLS connection. In an attempt to address these issues, we thoroughly analyzed a current SSL/TLS in the context of an SSL/TLS-based data transfer. More specifically, we closely investigated the dynamics of running an SSL/TLS-based Web interface and secure storage service within a real cloud computing platform [NexR-iCubeCloud 2011] by observing the following details:

(1) SSL/TLS buffer profiles in relation to the bandwidth of SSL/TLS connections and computation workload,
(2) SSL/TLS-based encoding/decoding and network processing routines,
(3) the memory allocation sequence with varying compression algorithms,
(4) the preceding three behaviors under severe network congestion and a bursty computation with varying levels of the computation power of the sender and receiver.

For this investigation, we narrowed down the security subsystems to an SSL/TLS protocol stack; this widely implemented stack is now the de facto standard for secure

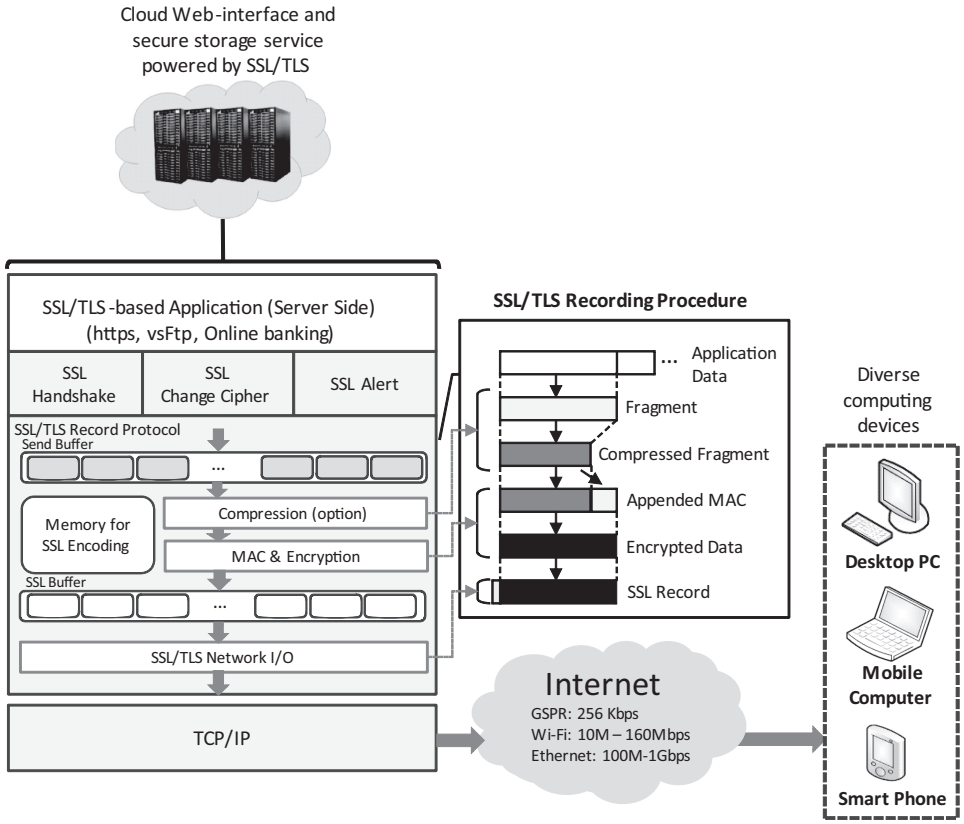

Fig. 1. Overall architecture of SSL/TLS with an expanded view of SSL/TLS protocol stack.

transactions over the Internet. Emerging cloud services are generally equipped with SSL/TLS for data confidentiality and integrity. SSL/TLS is designed to make use of TCP to provide a reliable end-to-end secure service. It offers an abstraction of secure sockets on top of existing TCP/IP sockets. As illustrated in Figure 1, SSL/TLS consists of two layers of protocols. The following higher-layer protocols are part of SSL/TLS: the SSL handshake, the SSL change cipher, and the SSL alert protocols. These protocols allow a server and client to authenticate each other, negotiate various algorithm parameters, dynamically update cipher suites, and send an alert message to a peer. The lower-layer protocol, namely the SSL/TLS record protocol, provides basic security services to various higher-layer protocols. For example, the cloud Web interface and secure storage services can operate on top of SSL/TLS. Data are sent via SSL/TLS connections in the following manner. First, an application writes data to an SSL/TLS send buffer. The record protocol then fragments the data into manageable blocks, optionally compresses the data, applies a message authentication code (MAC), encrypts the data, adds a header, and transmits the resulting unit to a receiver over the Internet. The received data are decrypted, verified, decompressed, reassembled, and then delivered to higher-level users. In this type of SSL/TLS-based transaction, the sender and receiver may have a huge computation power gap because the diverse computing devices can be connected with the server side. Furthermore, various connection speeds in terms of bandwidth and delay have values that differ in a given network environment. Our

analysis discloses three rudimentary operations of the SSL/TLS-based system. They are summarized as follows.

—*Underutilized computation and network resources of an SSL/TLS record protocol.* As the SSL/TLS segments were being processed, we observed underutilized computation and network resources on both the server and client side as a result of the sporadic computation and communication bottlenecks that occurred when the network and computation workloads were varied. In the event of underutilization of the given network and computation resources, the SSL/TLS throughput can be improved by maximizing the utilization of the given resources.
—*Static compression mode of SSL/TLS.* Applying a certain compression method to an SSL/TLS connection may not be optimal if the connection and computation workload are different and dynamic. If too much data are loaded for a low-bandwidth SSL/TLS connection, a compression mechanism should be cognitively applied with a reflection on environment heterogeneity (device, network, and cloud). However, because conventional SSL/TLS mechanisms provide a static compression mode, a renegotiation request by an application must change the compression algorithm to be applied. We therefore need to provide a mechanism that enables SSL/TLS to identify the best compression technique for SSL/TLS connections in a timely and transparent manner and in consideration of the characteristics just mentioned.
—*Obstructive compression switching overhead*. When the compression algorithm is changed frequently by the adaptive module, the compression switching overhead can cause a severe problem with transmission latency and thereby minimize the improvement. To deal with the compression switching overhead in SSL/TLS, we found that the memory allocation can be unified among several compression modules. However, current compression modules have a heterogeneous memory layout and are maintained by a separate compression library, which tends to duplicate the memory operations. Thus, we need a new mechanism that unifies the memory layout and shares the buffered data among the compression modules.

As a remedy to the three rudimentary operations of the SSL/TLS-based system, we designed and implemented ACCENT in OpenSSL 1.0.0 [OpenSSL.org 2011]. ACCENT is integrated into a Web interface layer and an SSL/TLS-based secure storage service within a real cloud computing service, called *iCubeCloud*, as the key primitive for SSL/TLS-based data delivery over the Internet. Specifically, we devised the following three mechanisms that improve the data transfer performance over SSL/TLS.

—*Tightly coupled threaded SSL/TLS coding (TTSC).* The purpose of TTSC is to maximize the computation and communication utilization when SSL/TLS data segments are sent and received. In conventional SSL/TLS, the computation routines, such as the compression and encryption operations and the SSL/TLS network routines are executed in a loosely coupled manner. This type of execution triggers frequent blocking and wake-up operations in the SSL/TLS process. As a remedy to this problem, our new mechanism provides the highest possible throughput by cleanly separating the computation routines from the network routines and by blending compressed and uncompressed packets in an SSL/TLS transmission.
—*Floating-scale based compression negotiation (FSCN).* The FSCN is a novel adaptive compression mechanism. It dynamically adjusts the compression algorithm by using floating scales of the sender and receiver, which move up and down in relation to the dynamic states of the encryption and compression data rate and the compression ratio. A floating scale consists of multiple scales. Each scale is defined in terms of the computation index (CI) of each encoding scheme, which reflects the computational characteristics. By finding the shortest distance between each scale and the current

network bandwidth, our new mechanism derives the optimal and most profitable SSL/TLS encoding method in a heuristic and transparent manner.

—*Unified memory allocation for compression (UMAC).* The purpose of UMAC is to minimize the compression switching overhead by unifying the memory footprints of the compression algorithms. Whenever a compression algorithm is changed, UMAC transforms the allocated memory layout to another layout. It either transforms the memory layout into a new memory layout of homogeneous compression algorithms or merely allocates the additional memory required by the upcoming compression algorithms. As a result, it significantly shortens the compression switching latency.

Our design is fully backwards compatible with SSL/TLS: extended clients can interact with servers that are unaware of our extensions, and vice versa. A client or server that fails to understand or prefers not to use these extensions can revert to standard SSL/TLS transactions. ACCENT is a part of our development project for a peta-scale cloud computing platform development project [NexR-iCubeCloud 2011; Park et al. 2010]. In our platform, ACCENT is integrated into the Web interface layer and SSL/TLS-based secure storage service within a real cloud computing service, called *iCubeCloud*, as the key primitive for SSL/TLS-based data delivery over the Internet. ACCENT can also be extended to cloud-based services such as a Web-based storage service and a content delivery service over SSL/TLS. It is also feasible for use with emerging applications such as a cloud-based mobile office and a cloud storage system. This type of facilitation is possible as long as the relevant applications utilize SSL/TLS as an underlying security layer. ACCENT is not targeted at small file transfers that seem unaffected by the dynamics of a network and computation workload; instead, it is targeted at large files and streaming data transfers.

The remainder of this article is organized as follows. In Section 2, we discuss previous studies on secure and efficient data delivery over the Internet. In Section 3, we elaborate ACCENT and its three novel mechanisms. In Section 4, we present our experimental results. Finally, in Section 5, we discuss our conclusions and future works.

2. RELATED WORKS

2.1. Previous Research on SSL/TLS-Based Systems

Many studies have analyzed and evaluated SSL/TLS-based systems for various kinds of computing devices and network environments [Coarfa et al. 2006; Zhao et al. 2005; Berbecaru 2005; Kant et al. 2000]. By profiling SSL/TLS processing, they tried to identify the performance bottlenecks and techniques for architectural improvement. They assert that the performance of SSL/TLS depends significantly on the network bandwidth, the computation power, and the hardware architecture. Those works became the motivation of this work.

For performance improvement of SSL/TLS-based systems, most studies have focused on accelerating individual network and computation routines or improving system performance from the perspective of the server alone rather than the perspective of both the client and server. Few studies have examined performance improvement as well as network and CPU utilization from a global consideration of the end-to-end network and computation status.

How to accelerate SSL/TLS computation by architectural optimizations or new hardware techniques has been the subject of several studies [Wu et al. 2001; Kounavis et al. 2010; Jang et al. 2011; Alaidaros et al. 2007; Morales-Sandoval and Feregrino-Uribe 2005; Chou 2002]. CryptoManiac [Wu et al. 2001; Kounavis et al. 2010] presents a coprocessor that combines arithmetic and logical operations for SSL/TLS performance improvement on traditional CPU architectures. Its added instruction set for fast cryptographic operations significantly improves SSL/TLS performance by means

of an additional functional unit. The crypto-acceleration units were added to the latest general-purpose Intel processors. In a recent study, SSLShader [Jang et al. 2011] presented a high-performance SSL acceleration technique using graphics cards as high-performance SSL accelerators. Other studies have focused on cryptographic algorithms and proposed optimizations for accelerating these crypto operations [Alaidaros et al. 2007; Morales-Sandoval and Feregrino-Uribe 2005; Chou 2002].

In an attempt to optimize the SSL/TLS transactions from the network perspective, seven research groups [Okamoto et al. 2003; Shacham et al. 2004; Castelluccia et al. 2006] proposed many techniques that improve the performance of SSL/TLS through compression or handshake optimization. SSLComp [Okamoto et al. 2003] is a general-purpose compression algorithm for SSL/TLS connections. It ignores abrupt changes included in SSL/TLS connections by the network bandwidth or computation workload. Other studies have analyzed the performance of SSL/TLS handshakes and suggested that improvements could be made by balancing the computation workload of the sever side or by caching the session data [Castelluccia et al. 2006; Shacham et al. 2004]. The SSLBalence alleviates the server load in the SSL/TLS handshakes and focuses on altering the computational balance between SSL clients and servers [Castelluccia et al. 2006]. CSSC is a caching mechanism for SSL/TLS handshake information on SSL/TLS clients; it reduces the network bandwidth consumption and boosts the SSL/TLS handshake speed [Shacham et al. 2004]. When ACCENT used in conjunction with SSLBalence and CSSC, the performance can be improved further.

ACCENT seamlessly adapts to heterogeneous environments (device, network, and cloud) and workloads so that it improves the computation and communication utilization to provide the highest possible throughput. It is capable of seamlessly applying the most profitable compression module for the characteristics of a given SSL/TLS connection and the current computation power of the sender and receiver. As a result, we believe that the complementary relationship between ACCENT and the previous works significantly improves the performance.

2.2. Previous Research on Efficient Network Systems

Several studies have proposed many mechanisms that improve the network throughput for multimedia systems, particularly with regard to novel encoding techniques [Haleem et al. 2007; Wu and Kuo 2005] and for general-purpose network systems [Krintz and Sucu 2006; Jeannot and Knutsson 2002; Wiseman et al. 2005]. In the research on video and multimedia systems, joint encryption and compression schemes for multimedia system are proposed [Haleem et al. 2007; Wu and Kuo 2005]. Although it can improve the computation efficiency by means of entropy coding and lossy compression for multimedia system, the mechanism is not suitable for protecting confidential data in a general-purpose system such as a SSL/TLS-based system. Lee et al. [2007] asserted that the quick adoption of AES and the use of a strong RSA key size of 1024 bits or higher are required in modern SSL/TLS-based systems.

For performance improvement of general-purpose network systems, researchers have studied adaptive network systems enhanced with compression algorithms [Krintz and Sucu 2006; Jeannot and Knutsson 2002; Wiseman et al. 2005]. Adoc [Jeannot and Knutsson 2002] and Wisen [Wiseman et al. 2005] presented an analysis of compression algorithms and their trade-offs and provided a well-defined compression interface as a middleware approach. Krintz and Sucu [2006] improved the Internet transfer performance by dynamically applying compression with the aid of prediction-based models supported by a mechanism that monitors the infrastructure and resource. This approach, however, is only applicable if the online prediction system is available.

Although our work is conceptually similar to the other works mentioned in this section, there is a key difference in the way we handle SSL/TLS computation and

network operations for the current network and computation resources of the two endpoints, which can seamlessly adapt to heterogeneous environments (device, network, and cloud) and workloads. We devised SSL/TLS coding mechanisms that cope with the abrupt changes caused in SSL/TLS connections by either the network bandwidth or the computation workload. Because a single SSL/TLS server services multiple and diverse SSL/TLS clients, careful attention should be paid to the current status of each SSL/TLS connection during server and client SSL/TLS transactions. We therefore focus on SSL/TLS encoding mechanisms for the end-to-end network and computation workload of the server and clients. In this way, we can ensure high-performance data delivery over an SSL/TLS system.

3. DESIGN OF ACCENT

As described in Section 1, in an attempt to achieve a cognitive SSL/TLS with heterogeneous environments and workloads awareness, we designed ACCENT to solve the limitations of conventional SSL/TLS-based systems. It has three novel mechanisms: TTSC, FSCN, and UMAC. Each mechanism is described in the following.

3.1. Tightly-Coupled Threaded SSL/TLS Coding (TTSC)

The objective of TTSC is to maximize the utilization of computation and communication resources to provide the highest possible throughput. The high throughput is achieved by cleanly separating the computation routine from the network routine and by blending the compression-enabled packets and compression-disabled packets in an SSL/TLS transmission. To integrate the routines without breaking any semantic of an SSL/TLS protocol stack, we need an in-depth understanding of the SSL/TLS protocol stack and the send and receive routines of the SSL/TLS network layer. These two issues as well as the TTSC are elaborated in the following.

The SSL/TLS protocol stack is the key component of a security subsystem. It consists of the following three components:

(1) SSL/TLS-packet processing routines, called a SSL/TLS record protocol; they include data segmentation, compression, encryption of packet, and the calculation of a MAC;
(2) SSL/TLS buffer management, which includes a send and receive SSL/TLS buffer;
(3) protocols for an SSL handshake, an SSL change cipher, and an SSL alert.

Due to the nature of SSL/TLS, all SSL/TLS packets sent to clients are processed by the SSL/TLS-packet processing routines just mentioned. The packet transfer rate and latency of each SSL/TLS session vary in relation to the computation power of the client and the server and the dynamic status of the network path established between the server and the client. From the perspective of a TCP/IP protocol stack, an SSL/TLS subsystem is a data provider whose role is to encode an adequate amount of data from an application's buffer into TCP send buffers. This type of encoding is based on consideration of the transmission rate of a corresponding SSL/TLS connection.

In the following, we explain how conventional SSL/TLS is implemented in a widely-deployed SSL/TLS protocol such as OpenSSL. The execution routine for sending SSL/TLS data occurs in the order shown in Figure 2.

(1) *Fragmentation.* An SSL/TLS record protocol works by fragmenting the data to be transmitted into a series of fragments, each of which is independently protected and transmitted. Depending on the SSL/TLS implementation, the protocol may copy the data into buffers in a user space.
(2) *Compression.* Each fragment is compressed provided compression options are enabled and stored in a temporary buffer.

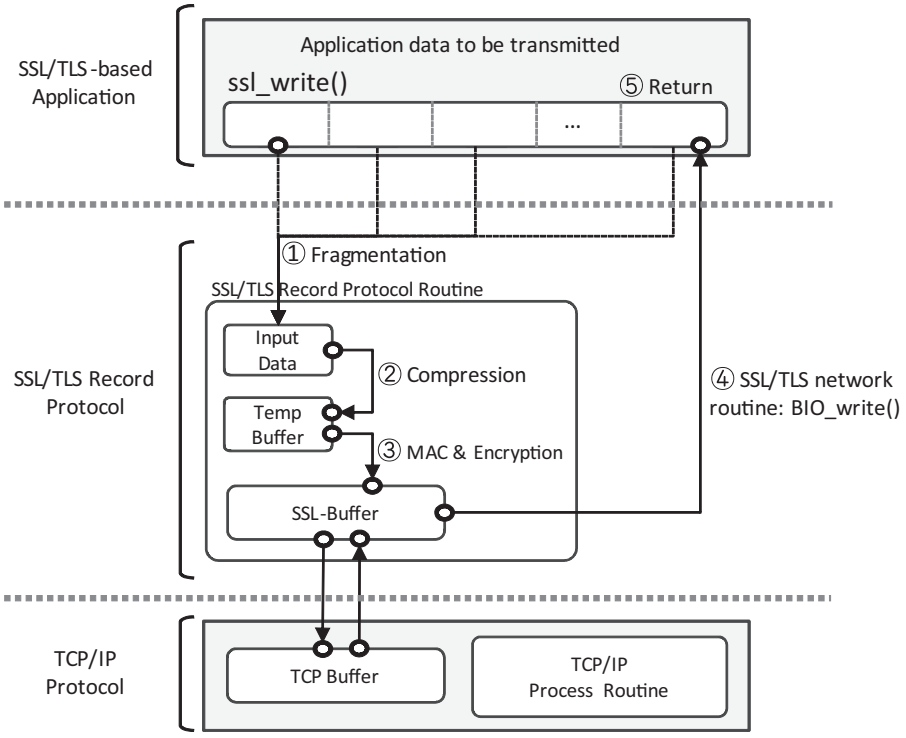

Fig. 2. Execution routines for sending SSL/TLS data of conventional SSL/TLS.

(3) *MAC and encryption*. The next step involves the computation of a MAC, which facilitates message integrity. The compressed message and MAC are encrypted, and a header is then attached to the payload to complete the assembly of the SSL record.
(4) *SSL/TLS network routine*. The encoded data are stored in an SSL buffer and an SSL/TLS network routine is called. The routine is then blocked in the blocking I/O until the corresponding acknowledgement segments are transmitted. The blocking may occur very frequently due to the mismatch between the data generation rate of the SSL/TLS record protocol and the packet consumption rate of the TCP.

As explained, whenever SSL/TLS sends data to a client, the send routine should be blocked until the corresponding acknowledgement segments are transmitted [Rescorla 2001]. As a result, the computation resource for processing the SSL/TLS segments in both server and client side tend to be underutilized. Thus, the current SSL/TLS obviously suffers from the frequent blocking operations.

To see how the bandwidth of an SSL/TLS connection affects the utilization of computation and communication, we set up an experimental environment comprised of a client, a server, and a WAN router [Carson and Santay 2003]. The server and client had identical hardware specifications (CPU: Xeon E5550, RAM: 4GB) which is a front-end Web interface server of our cloud computing platform. The router emulates a WAN environment by providing for each connection a set of configurable parameters, such as the latency and bandwidth. We established a hundred SSL/TLS connections between the server and the client via the router with various connection bandwidths. When a connection became stable, the effective throughput was probed. We note that

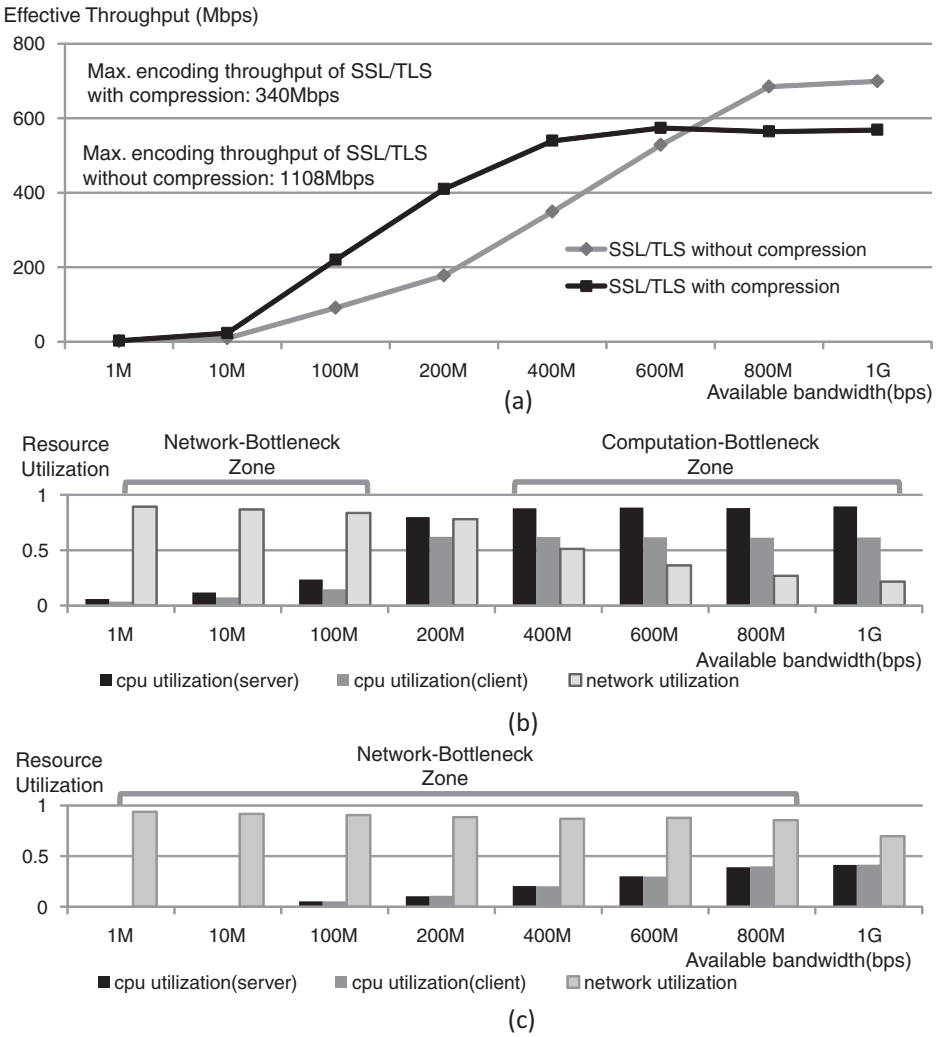

Fig. 3. Effective throughput and CPU/network utilization with various available bandwidths: (a) effective throughput, (b) resource utilization of compression (lzo)-enabled SSL/TLS, and (c) resource utilization of compression-disabled SSL/TLS.

the effective throughput is the throughput considering the compression effect. In this experiment, we applied lzo [Oberhumer 2010] as a compression module of the SSL/TLS.

From Figure 3, we can summarize the following observations.

—*Trade-off analysis of compression-enabled and compression-disabled SSL/TLS recording strategies.* As the available bandwidth increases, the effective throughput (regarding the compression effect) of the compression-enabled and compression-disabled SSL/TLS connections also increase with different incline. Figure 3(a) shows the saturated effective throughput of the compression-enabled and compression-disabled SSL/TLS at available bandwidths of about 600Mbps and 800Mbps, respectively. In the case of the compression-enabled SSL/TLS, even when the available network bandwidth exceeds the SSL/TLS encoding throughput (340Mbps), some regions are guaranteed to win the compression-disabled SSL/TLS effective throughput. This

behavior occurs because the compression-induced gain exceeds the underutilized network bandwidth. When the available network bandwidth sufficiently exceeds the SSL/TLS encoding throughput, it cannot outperform the compression-disabled SSL/TLS effective throughput since it starts to underutilize the network bandwidth that exceeds the compression-induced gain.

—*Computation and communication underutilization by different types of bottlenecks.* As shown in Figure 3(b), at the low end of the available bandwidth (1Mbps to 100Mbps), the compression-enabled SSL/TLS connection is on the bottleneck zone of the network, that is, where the network bandwidth is fully utilized but the computation resource is underutilized. This situation remains the same as long as the available network bandwidth cannot sufficiently transmit all of the compressed data. The bottleneck zone of the network is delimitated by a given SSL/TLS encoding throughput (340Mbps), the point at which the compression thoroughly utilizes the CPU and the compressed data fully occupies the available network bandwidth. At the high end of the available bandwidth (400Mbps to 1Gbps), the compression-enabled SSL/TLS connection is on the bottleneck zone of the computation. It saturates the CPU and underutilizes the network bandwidth. Conversely, as shown in Figure 3(c), the compression-disabled SSL/TLS encoding saturates the network bandwidth but leaves the CPU underutilized in the overall region. As a result, it is on the bottleneck-zone of the network because the SSL/TLS encoding throughput sufficiently exceeds the available network bandwidth.

Static SSL/TLS encoding obviously causes inefficient communication and computation utilization. In addition, the static SSL/TLS encoding strategy causes additional inefficiency of the computation and communication resources in data-intensive servers or devices with low computing power [Qi et al. 2009]. Therefore, to maximize the utilization of the CPU and network bandwidth, we developed TTSC, which separates the computation routine from the network routine and blends compressed and uncompressed packets regarding the data transmission rate of a corresponding SSL/TLS connection. Therefore, deciding the amount of data to be compressed for the performance improvement of SSL/TLS is a key problem in TTSC. Prior to illustrating how TTSC works, we need to explain how the states of SSL/TLS connections are maintained in an SSL/TLS protocol stack. Dynamic SSL/TLS states can be obtained at a given moment by probing the variables of an SSL/TLS buffer that is maintained by an SSL/TLS protocol stack [Rescorla 2001].

—*ssl_buffer.left* is the size of data to be sent to the TCP.
—*ssl_buffer.offset* is the address pointer of the next data byte to be sent.

The left edge of the SSL/TLS buffer in Figure 4 is *ssl_buffer.offset*; the right edge is *ssl_buffer.left*. Whenever an SSL/TLS process receives an acknowledgement segment, the pointer *ssl_buffer.offset* moves to the right along the space of the SSL/TLS buffer address. The relative interval of the two edges of the buffer either increases or decreases its available buffer space for outgoing data. The speed of the pointer movement is mainly determined by the bandwidth of the SSL/TLS connection. With that information, we can estimate the approximate transmission data rate of a given SSL/TLS connection.

To devise our new SSL/TLS encoding mechanism from scratch, TTSC is based on the transmission data rate (TD). A TTSC monitor is used to measure the new state variables needed to observe the status of the SSL/TLS encoding. The variables are defined in TTSC for an individual SSL/TLS connection as follows.

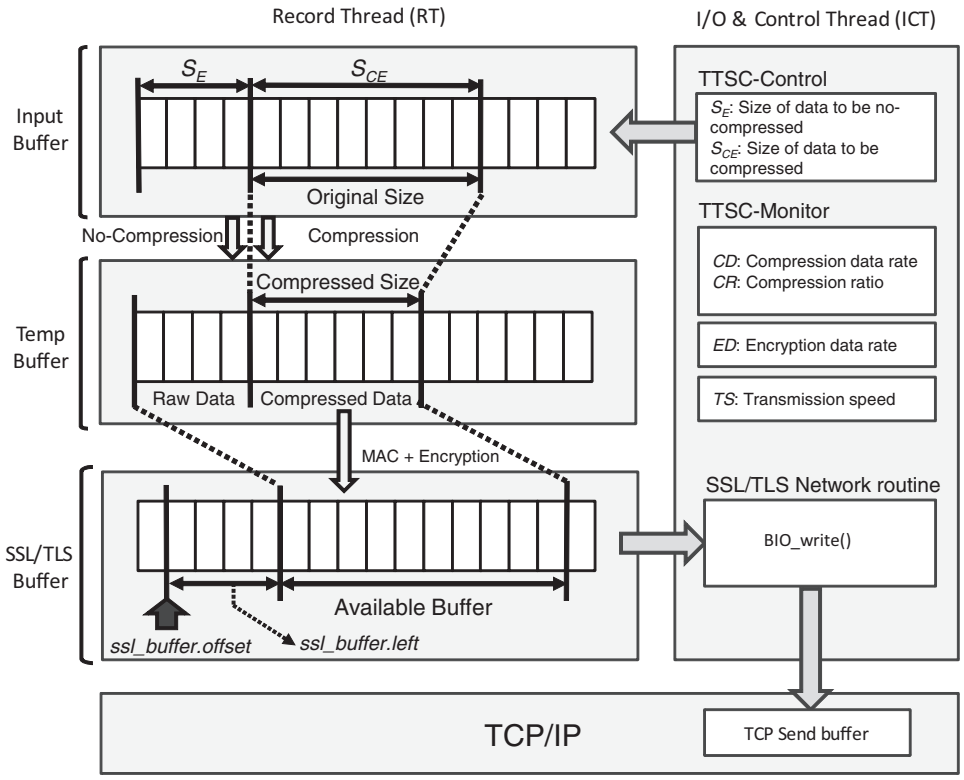

Fig. 4. SSL/TLS buffer space and variables for TTSC-based SSL/TLS send/receive routines.

—Encryption data rate (ED). This variable is the average data rate of an encryption in an interval of time; it is expressed as the generated amount by the encryption divided by the duration of the interval.
—Compression data rate (CD). This variable is the average data rate of a compression in an interval of time; it is expressed as the generated amount by the compression divided by the duration of the interval.
—Compression ratio (CR). This variable is a value that represents the ratio of the compressed data length to its original data size. The ratio is calculated by Compressed Size / Original Size.

To measure the size of data to be compressed, S_{CE}, we consider the monitored values of TD, ED, CD, and CR.

Definition 1. The alpha value of an SSL/TLS connection is defined as the time ratio T_{comp}/T_{total} for the connection, where T_{total} is the total encoding time and T_{comp} is the time of the data to be compressed.

Alpha represents the potential network saving and computational overhead. A larger α value indicates that the TTSC technique achieves a greater network traffic saving and computation utilization. When the monitored values of TD, ED, CD, and CR are obtained, we can calculate the optimal ratio, α, for a given SSL/TLS connection by considering TD to be equal to the data rate for the compression plus the encryption with the optimal ratio, α.

In this equation,

$$\frac{CD \cdot ED}{CR \cdot CD + ED} \cdot \alpha + ED \cdot (1 - \alpha) = TD,$$

$$\alpha = \begin{cases} 0, & \text{if } ED < TD \text{ and } CD < TD \\ \dfrac{TD \cdot CR - ED}{CD/(CR \cdot CD + ED) - ED} & \\ 1.0, & \text{if } CS > TD, \end{cases} \quad (1)$$

$CD \cdot ED/(CR \cdot CD + ED)$ is a derived compression-encryption data rate considering the current CR. ED is an encryption data rate and TD represents the transmission data rate. α is a coefficient that aligns the encoding data rate into the TD by partially applying compression with the ratio, α. TTSC needs to ensure that α satisfies Equation (1) to sustain the outgoing bandwidth of the given SSL/TLS connection. Once α is obtained, the amount of data to be compressed (S_{CE}) and the amount of data to be left uncompressed (S_E) can be calculated as follows:

$$S_{CE} = \begin{cases} \dfrac{\alpha \cdot S \cdot CD}{TD}, & \text{if } 0 < \alpha < 1.0 \\ S, & \text{if } \alpha > 1.0 \end{cases} \quad (2)$$

$$S_E = S - S_{CE},$$

where S is the total size of the data from the application layer. Whenever the data from an application layer arrives at an SSL/TLS record layer, TTSC encodes S_E Kbytes to the compression-disabled SSL/TLS segment and S_{CE} Kbytes to the compression-enabled SSL/TLS segment to maximize the utilization of the CPU and the network.

3.1.1. Implementation Details in OpenSSL. This section illustrates how to implement TTSC in OpenSSL 1.0.0. Figure 5 shows the detail of the data delivery over SSL/TLS enhanced with TTSC at a function level. In TTSC, a set of routines executed in one thread (SSL/TLS-based data sending routines of Figure 2) is integrated into two threads, a record thread (RT) and an I/O and control thread (ICT) as shown in Figure 5. Once the size of the data to be compressed (S_{CE}) and the size of the data to be left uncompressed (S_E) are derived in this scheme by the ICT, the RT compresses and encrypts packets according to the derived values and the ICT performs SSL/TLS network routines. The two threads interact through an SSL/TLS buffer. The RT takes a certain amount of data from the application layer and puts encoded packets into the SSL/TLS buffer after the encoding. The ICT takes the encoded packets from the SSL/TLS buffer and sends them over the TCP. By virtue of the threaded encoding, which cleanly separates the computation routines from the network I/O routines, TTSC can continue to supply a proper amount of encoded data to the SSL/TLS buffer with improved network and computation utilization.

3.2. Floating-Scale-Based Compression Negotiation (FSCN)

To transfer data to a client over SSL/TLS, we first need to encode the data by means of compression and encryption algorithms and load them into an SSL/TLS buffer. The compression technique increases the amount of bandwidth available to a given SSL/TLS connection. By reducing the amount of data transferred, we increase the

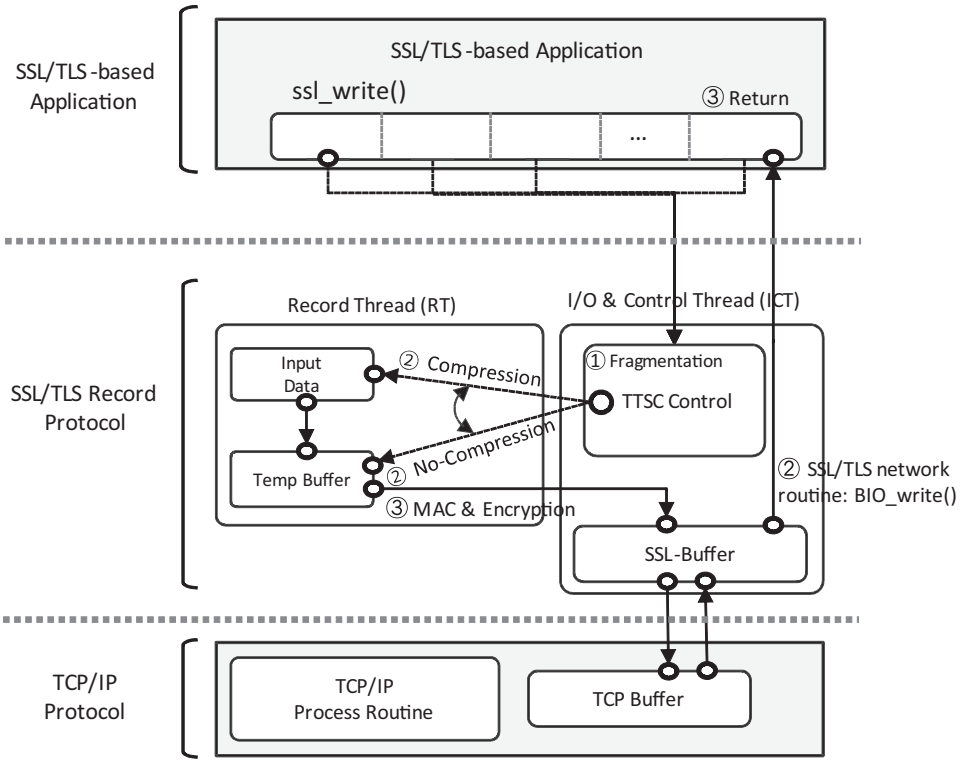

Fig. 5. Execution routines for sending SSL/TLS data of TTSC-based SSL/TLS data transfer.

available bandwidth and reduce the transmission time. However, compression involves the following challenging issues for efficient SSL/TLS communication.

—First, compression techniques vary from a performance perspective, particularly with regard to the compression ratio (CR), compression time, and decompression time. Techniques are either optimized for the CR or execution time. The process consequently involves a trade-off between various factors, including the algorithmic complexity and the CR. Thus, applying a certain compression method to an SSL/TLS connection may not be optimal.
—Second, the compression performance relies on the availability and performance of the underlying resources, such as the network bandwidth, the latency, and the computation power. This performance varies significantly across environments; it also varies over time for the same environment. The latter impacts mobile devices such as smartphones, tablets, netbooks, and notebooks. For mobile devices, the available underlying networking technology changes regularly [Brik et al. 2005], for example, network switching occurs in a given network infrastructure, such as GPRS, Wi-Fi, and 100Mb/1Gb Ethernet.

Motivated by these observations, we devised a novel compression negotiation mechanism, FSCN, to enable a form of negotiation that automatically and transparently identifies and applies the best compression technique where the connection and computation workload are different and dynamic.

In our implementation, we chose run-length encoding (RLE) [Geelnard 2006], lzo [Oberhumer 2010], gzip [Gailly 2010], and bzip [Seward 2010] as the set of

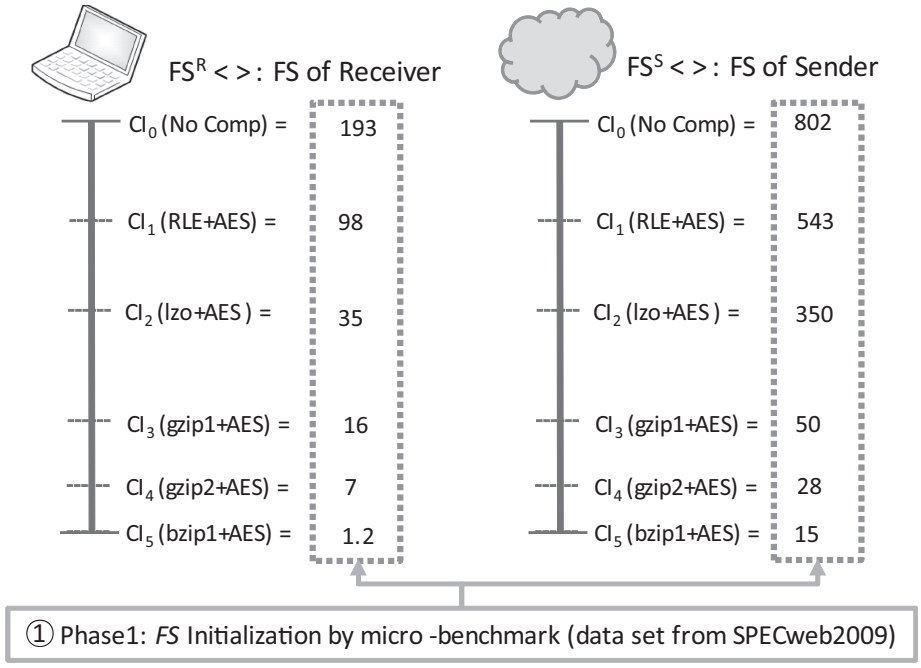

Fig. 6. Defined floating scale consists of a set of profiled computation index (CI).

compression modules of SSL/TLS. The reasons for this selection are summarized as follows. First, the selected compression algorithms have different performance characteristics in terms of the compression ratio and the compression/decompression time. The bzip is optimized for the compression ratio at the cost of the compression and decompression times. RLE and lzo have much faster compression and decompression times but a low compression ratio, lower than bzip. In the case of gzip, the performance characteristics in terms of the compression ratio and compression/decompression time fall between those of LZO and bzip. The set of compression modules with different performance characteristics provides an opportunity to apply the most profitable compression module cognitively considering the given SSL/TLS connection and the current computation power of the environmental heterogeneity of the sender and receiver. Second, the compression algorithms are selected because of their wide availability and practicality. The compression algorithms (RLE, lzo, gzip, and bzip) commonly provide open-source libraries with well-defined interfaces. This makes it possible to integrate them into the SSL/TLS protocol stack as a set of compression modules.

The core component of FSCN is the floating scales (FS) of the sender and receiver because FSCN uses the FS to dynamically adjust the compression algorithm. As shown in Figure 6, the FS consists of multiple scales. Each scale is mapped to the computation index (CI) of each encoding scheme (compression algorithm plus encryption algorithm); the CI reflects the computational characteristics, such as the data rate for the compression plus the encryption and CR. During the installation phase of ACCENT, each CI value is locally derived and updated into the FS by means of a micro-benchmark. From the results of the micro-benchmark, the value of each CI can be measured and initialized into the FS so that the initial FS is constructed (*Phase 1: Initialization*). After that, each CI of the sender and receiver is periodically monitored and updated into the FS through a real SSL/TLS transmission so that the FS moves up and down

in accordance with the dynamic states of the underlying computation power, and CR (*Phase 2: FS Evaluation*). Hence, the FS of the sender and receiver keeps track of the CI_i associated with each encoding scheme, i. In addition, the FS values of the sender and receiver are exchanged with each other (*Phase 3: Exchange*). Finally, by finding the shortest Euclidean distance between each CI and the current network bandwidth, our new mechanism heuristically determines the most profitable compression method (*Phase 4: Negotiation*). Because FSCN aims to provide an adaptive SSL/TLS encoding mechanism, any kind of encoding methods included in the FS can be applied to a given SSL/TLS connection, ensuring that the SSL/TLS connection performs the transmission at the highest possible throughput.

In summary, the FSCN process has four phases. Phase 1 runs on the server and client side to initialize the FS in the installation phase of ACCENT. Phase 2 periodically updates the FS of the sender and receiver. Phase 3 exchanges the derived FS between the sender and receiver. Phase 4 conducts a negotiation on the selection and application of an encoding scheme. Whenever FSCN is enabled, the following four phases are performed to determine the optimal encoding scheme for the sender and receiver.

Phase 1 (*Initialization*). This phase, which is part of the installation of ACCENT, is an initial profiling phase. Each CI value of the FS is locally derived and updated by means of a micro-benchmark. The micro-benchmark performs SSL/TLS recording routines for each encoding scheme; for this purpose, it uses a training set consisting of a hundred 16KB blocks from a well-defined Web traffic dataset from SPECweb2009 [SPEC 2010]. The micro-benchmark results indicate the CR, CD, and ED. After that, each CI can be derived as follows:

$$CI_\delta = \frac{CD_\delta \cdot ED}{CR \cdot CD_\delta + ED} + CD_\delta(1 - CR),$$

(1) Result from Micro-benchmark
  CD: Compression Date Rate
  CR: Compression Ratio
  ED: Encryption Data Rate
(2) First term: Encryption-compression data rate
$$= \frac{\text{Total Size}(S)}{\text{Total Time}} = \frac{S}{S/CD + S \cdot CR/ED} = \frac{CD_\delta \cdot ED}{CR \cdot CD_\delta + ED}$$
(3) Second term: Reduction-speed
$$= \frac{\text{Reduced Size}}{\text{Total Time}} = \frac{S - CR \cdot S}{S/CD} = CD(1 - CR),$$

(3)

where δ is the currently applied encoding scheme, and the first term, $CD_\delta \cdot ED/(CR \cdot CD_\delta + ED)$, is the derived data rate for the compression plus the encryption, and the second term, $CD_\delta \cdot (1 - CR)$, is the data reduction speed derived from the corresponding compression module. Therefore, Equation (3) reflects the potential data rate with consideration of the computation overhead and the data reduction speed of a certain compression method. Each CI value can be obtained by applying Equation (3) to each encoding scheme. The CI values are organized into a type of array ($FS <>$), and stored locally as shown in Figure 6. Finally, FSCN locally accepts a set of profiled CI values as input to construct the initial FS. In addition, Phase 1 can be performed again whenever the hardware is changed or a new compression method is added.

Phase 2 (*FS Evaluation*). FSCN considers the underlying network status, the computation power, and the CR of each encoding scheme as part of the FS estimations. In this phase, the values of FS of the sender and receiver are periodically monitored and

ALGORITHM 1: Updating the CI^S and CI^R of the current encoding scheme

Input: Input parameters of Equation (3) to derive CI_δ
Output: Updated FS^S <> and FS^R <>

1 //CI^S : CI value of sender;
2 //CI^R : CI value of receiver;
3 //CI_δ : CI of current encoding scheme;
4 //$CI_{\delta(-0)}$: the latest CI of current encoding scheme;
5 //$CI_{\delta(-\omega)}$: CI in the ω-th previous time quantum;
6 invoke performance monitoring framework;
7 derive CI_δ^{-0} using **Equation (3)**;
8 $CI_\delta^{avg} \leftarrow$ get average of (CI_δ^{-w} to CI_δ^{-0});
9 $difference \leftarrow CI_\delta^{avg} - CI_\delta^{-0}$;
10 **foreach** *CI element from* FS^S <>, FS^R <> **do**
11 | sender: $CI^S \leftarrow FS^S$ <*index*> + *difference*;
12 | receiver: $CI^R \leftarrow FS^R$ <*index*> + *difference*;
13 **end**
14 set an update timer to expire after q sec;

updated through a real SSL/TLS transmission so that the FS moves up and down in accordance with the dynamic states of a given computation power, and CR value.

The overall procedure of Phase 2 is described as follows in Algorithm 1, where FS^S <> and FS^R <> are the FS of the sender and receiver, respectively. Since the initialized FS by Phase 1 can be changed by a current computation workload and CR, this update is based on the process speed and CR of the current SSL/TLS transmission during the w previous time quanta. Despite the algorithmic differences, we assume an approximately linear relation among the different compression methods in terms of the CR and process speed [Krintz and Sucu 2006]. The FS can therefore be updated on the basis of the CI of the current encoding scheme.

Algorithm 1 updates FS^S and FS^R before every time quantum. This enables each FSCN of the sender and receiver to keep track of the CI of the current encoding scheme, $CI_{\delta(-0)}$ (lines 6 and 7 of Algorithm 1). The notation $CI_{\delta(-\omega)}$ denotes the CI in the wth previous time quantum. The runtime parameter, ω, defines the sliding window over which the derived CI values are averaged (lines 8 of Algorithm 1). The average value of ω previous CI values can be used to derive the difference (line 9 of Algorithm 1). The FS^S and FS^R values are then updated so that the difference is reflected in each CI value of the FS (lines 10 to 13 of Algorithm 1). The FS^S and FS^R values are updated before every time quantum (line 14 of Algorithm 1). The choice of ω can affect the responsiveness of FSCN: When the value of ω is increased, short spikes of change are disregarded; When the value of ω is decreased, there is a quick adaptation to changes in the underlying environment.

Phase 3 (*Exchange*). The evaluation of FS^S and FS^R in Phase 2 is designed to be executed locally at the sender and receiver. Thus, the local values of the sender and receiver should be shared periodically with each other to ensure the proper selection of the compression modules. The FS^S and FS^R derived in Phase 2 are exchanged over the SSL/TLS packet header in a compact 1-byte format.

Phase 4 (*Negotiation*). In the final phase of FSCN, a negotiation operation takes place. For the FSCN-based negotiation, time is divided in quanta of q sec. At the end of

ALGORITHM 2: Selecting the encoding scheme for the next time quanta

Input: measured TD, FS^S <>, and FS^R <>
Output: selected encoding scheme (CI_{best})

```
1  // bestd: the shortest Euclidean distance so far;
2  // best: index number of bestd;
3  // d: measured Euclidean distance;
4  invoke FS of two end points into FS^S <> and FS^R <>;
5  TD ← measured TD;
6  bestd ← ∞; best ← −1;
7  foreach CI pairs from FS^S <>, FS^R <> do
8  |   CI^S ← FS^S <index>, CI^R ← FS^R <index>;
9  |   d ← distance(CI^S, CI^R, TD);
10 |   if d < bestd then
11 |   |   bestd ← d;
12 |   |   best ← index;
13 |   end
14 end
15 selected encoding scheme ← CI_best;
```

every time quantum, FSCN may change the compression module that is to be applied during the next time slots. The procedure for negotiating the local set of compression modules is described in Algorithm 2.

Algorithm 2 heuristically selects the encoding scheme. It estimates the fitness between the resource demands of each encoding scheme and the currently available resource, such as the computation power and the network bandwidth. The fitness can be measured by the sum of the distance between CI^S and TD, and the distance between CI^R and TD (lines 8 to 9 of Algorithm 2). The shortest distance is the main criterion for selecting the encoding scheme (lines 10 to 13 of Algorithm 2).

Figure 7(a) shows FSCN-based negotiation. This type of negotiation is based on the assumption that mobile devices such as smartphones and netbooks are connected to a server over SSL/TLS. In this case, the available underlying network interface changes regularly, for example, where a user uses Wi-Fi while attending a conference (State1); takes the device to work and connects via a 100Mb/1Gb Ethernet link (State2); and, while outdoors, connects the device to a GPRS network (State3). In FSCN, the encoding scheme for a given SSL/TLS connection is selected heuristically in an effort to balance the load of the computation overhead and the current network status; it does so by ensuring that the sender and receiver are neither saturated nor underutilized. Algorithm 2 selects the most fitting encoding scheme considering which scheme has the optimum resource consumption for the current TD (lines 7 to 13 of Algorithm 2). As shown in Figure 7(b), if we visualize the 2-D distribution of the CI values of each encoding scheme of the sender and receiver on the x axis and the y axis, respectively, we can derive the Euclidean distance between each CI and TD. The selection can be achieved by finding the CI with the shortest Euclidean distance among the invariant candidate CIs by following equation.

$$\text{Selected encoding scheme}(n) = \min_{n \in CI^S, CI^R} \left(\sqrt{(TS - CI^S)^2 + (TS - CI^R)^2} \right), \tag{4}$$

where n is the selected encoding scheme, which is a good fitness heuristic.

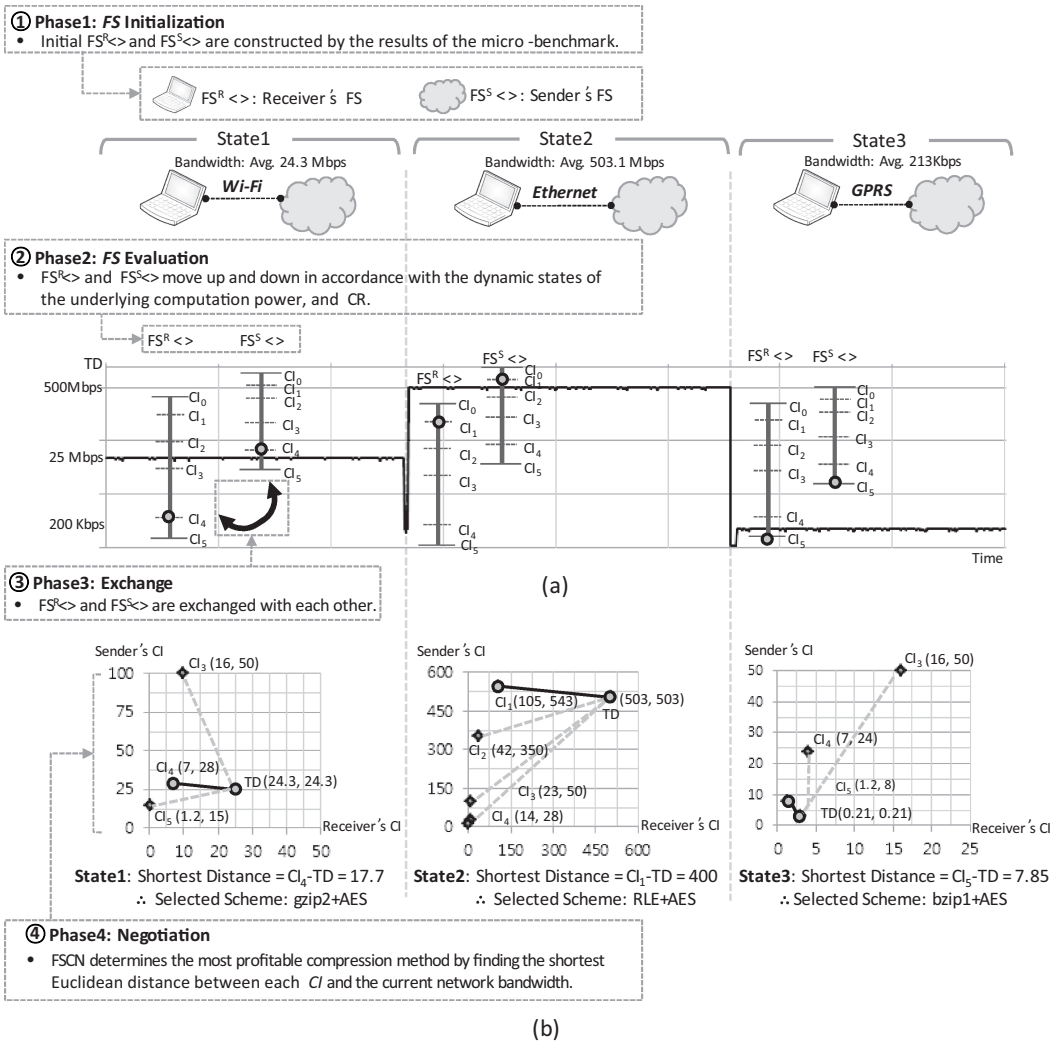

Fig. 7. FSCN-based negotiation under a scenario of various network and computation workloads (a) FSCN-based negotiation for a changing transmission data rate; (b) 2-D distribution of CI of each encoding scheme of the sender and receiver.

In the case of State1, the derived encoding scheme is CI_4 (gzip2 + AES), because the shortest Euclidean distance is between CI_4 (x: FS^R <4>=7, y: FS^S <4>=28) and the current TD (x: TD=24.3, y: TD=24.3). Depending on changes in the available network interface, the derived encoding scheme by FSCN is CI_1 (RLE + AES) for State2 and CI_5 (Bzip1 + AES) for State3. The use of this metric facilitates the algorithm selection by finding the shortest distance between TD and CI. Once the compression algorithm has been selected, the selected compression module is applied to the SSL/TLS data stream.

3.3. Unified Memory Allocation for Compression (UMAC)

In this section, we describe the solution for the most effective reaction of a memory allocator when the compression algorithm is changed frequently by the adaptive module.

Table I. Memory-Related Operation Profile and Compression Switching Overhead

Algorithms	Initial Overhead	Release Overhead	Consumed Memory
gzip	0.46 ms	0.24 ms	141.7 KB
bzip	1.31 ms	0.68 ms	453.28 KB
lzo	0.19 ms	0.08 ms	64 KB
RLE	0.15 ms	0.07 ms	16 KB

Note: Allocated memory size of the server side for compression and the memory initialization/reclamation time.

Table II. SSL/TLS Transmission Profile for One Packet Transmission Accompanying Compression Switch

	https	SSL/TLS			TCP/IP
SSL-Recording Scheme	get(), post()	malloc()	do_comp()	do_enc()	send(), recv()
gzip-AES	2.04%	7.75%	81.28%	3.13%	5.79%
bzip-AES	1.14%	14.19%	80.43%	1.48%	2.75%
lzo-AES	6.29%	8.81%	42.67%	14.81%	27.42%
RLE-AES	5.85%	7.03%	17.84%	24.29%	44.98%

Memory operations can be a burden because they generally proceed at memory speed rather than CPU speed and thereby slow the performance of the system [de Bruijn and Bos 2008]. Conventional SSL/TLS systems provide inadequate support for the adaptive module. As a result, they cause severe transmission latency, which hinders the improvement of the FSCN. This problem may be due to the memory footprint and the lack of integration among the various compression modules. Each compression module uses its own internal buffering and dictionary mechanism. Whenever the applicable compression module is changed, it releases the memory allocated for the dictionary and data buffering of the corresponding compression modules and it immediately terminates the compression operation. This approach leads to frequent bouts of memory allocation and release as well as other performance-degrading anomalies. This design has an obvious drawback, that is, the repeated memory allocation causes a high CPU overhead and limits the throughput of the SSL/TLS system.

In an attempt to fully understand the switching of the SSL/TLS compression module, we profiled a running https server, focusing on the memory allocation and reclamation of compression induced by Oprofile [Levon and Elie 2011]. Oprofile can collect execution profiles of all the execution entities of the (thread-level) processes and memory allocations. We conducted the profiling on the https server because it covers all of the SSL/TLS routines; an https transaction on the server is a superset of the SSL/TLS routines. For profiling, we interconnected a server (CPU: Xeon E5500, RAM: 4GB) and clients (CPU: Z530, RAM: 1 GB) with a dynamically changing network environment and computation workload to force the switching of the compression module. The clients established SSL/TLS connections with the server and then downloaded and uploaded diverse chunk files from the server as fast as possible.

Table I shows the allocated memory size of the server side for the compression and the memory initialization and reclamation time of each SSL/TLS connection. It also shows the duration of the switching algorithms. Each compression algorithm requires 16KB to 453.3KB of memory for the compression buffer and dictionary. Table II shows the profiling results of the server side for a packet transmission that accompanies the compression switch. The symbols in boldface represent the following entities: https, secure socket layer (SSL/TLS), and the TCP/IP. Around 7.03% to 14.19% of the CPU cycles are consumed in the various types of compression switching, such as data copying and memory allocation. In addition, TCP/IP takes 2.75% to 44.98% for protocol processing. On the other hand, https takes only 1.14% to 6.29% of the CPU cycles. The http protocol processing makes up only a small fraction of our experiments because a

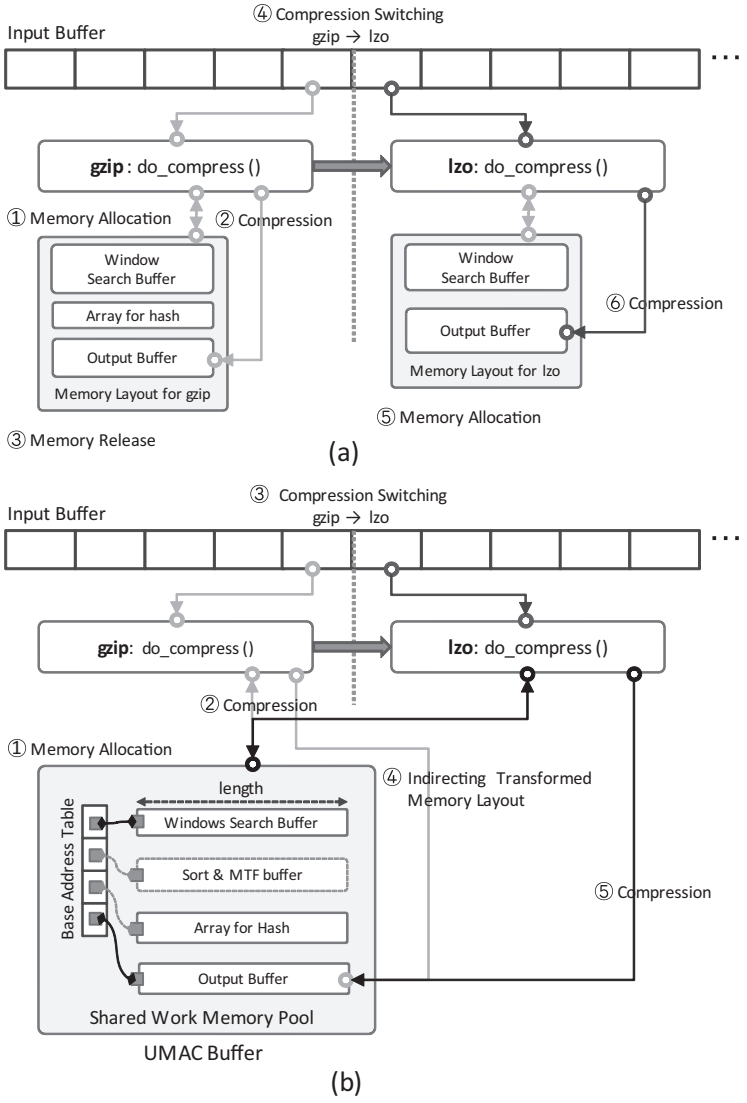

Fig. 8. Memory operations and compression switching. (a) Conventional memory allocation (b) UMAC-based memory allocation.

single SSL/TLS connection generates only one https request to download a chunk file. The profiling results suggest that compression switching can be burdensome when the compression algorithm is changed frequently.

Figure 8 shows the memory operations for compression switching in the SSL/TLS. Let us assume that a compression algorithm is switched from gzip to lzo as pictured in Figure 8(a). Whenever the compression module is changed, the current compression module releases the assigned memory, decreasing the transmission rate of the corresponding connection. At that moment, a new memory footprint is issued to a new compression module, which tends to duplicate the memory operations.

We devised UMAC to deal with the frequent compression switching in SSL/TLS. The UMAC is a technique to speed up the transition between compression algorithms

during an ongoing data transfer. It unifies the compression data buffers inside a shared memory instance. To achieve this, UMAC introduces a unified memory allocation structure. Figure 8(b) illustrates UMAC with the unified memory allocation structure. The structure consists of the base address table and the shared work memory pool for the compression modules. It is allocated in a region of the heap space called the UMAC buffer. The shared work memory pool can be commonly utilized for each compression module. In the case of gzip or lzo, the shared work memory pool is used to search buffers; in the case of bzip, it can be used to sort buffers. All compression modules can utilize the region through the unified memory layout of UMAC. The address of the shared memory region is passed among the compression modules in terms of the values (the start address and length) of the base address table. In contrast, the associated UMAC buffers are passed among the compression modules in terms of reference. This approach enables a single memory allocation for multiple compression modules to be shared throughout the corresponding SSL/TLS connection. This occurs because the UMAC replaces memory operations such as a memory allocation and release with the memory address passing for a new compression.

We note that UMAC does not require any modification of system-level functions such as *malloc()* or *free()*. The modification induced by UMAC involves the memory operation routines inside the compression library of OpenSSL. Consequently, UMAC is applicable not only to the specified compression modules but also to other compression schemes. Once the data structure for a certain compression scheme is registered in the unified memory layout of UMAC, a single instance of memory allocation for multiple compression modules can be shared among the other compression modules. This therefore significantly shortens the compression switching latency.

In brief, UMAC enhances the efficiency of memory operations, such as the memory allocation and release, by unifying the memory allocated to SSL/TLS connections when the compression switching occurs.

4. PERFORMANCE

In this section, we present the performance results obtained with our prototype version of ACCENT implemented in OpenSSL 1.0.0. It was developed as an internal code patch for core SSL/TLS routines without any modification to the current syntax of native SSL/TLS routines. Hence, no application modification or recompilation is required to utilize ACCENT as long as applications use SSL/TLS routines as a data transfer primitive. We perform the experiment within a real cloud computing platform [NexR-iCubeCloud 2011] running SSL/TLS-based Web interface and secure storage service on account of its practicality. The SSL/TLS used in our experiments is the open source OpenSSL 1.0.0. The https traffic load was generated with SPECweb2009, which is the industry standard for evaluating an SSL/TLS-based Web system performance [SPEC 2010]. As shown in Figure 9, the SSL/TLS-based Web interface of the cloud side has a Xeon E5500 processor and 4GB main memory. Four kinds of client machines with diverse hardware specifications, from handheld devices to the server, were used to drive the server with a variety of access patterns. To emulate a WAN environment, we used a NIST-WAN router that provides each connection with a set of configurable parameters, such as latency and bandwidth [Carson and Santay 2003]. All the machines were interconnected via a gigabit Ethernet. ACCENT can improve the performance of servers that handle a number of concurrent SSL/TLS connections; it can also improve the performance of clients. Therefore, the performance metrics of our experiments are the overall transmission rate of the server and client and the resource utilization under various workload patterns.

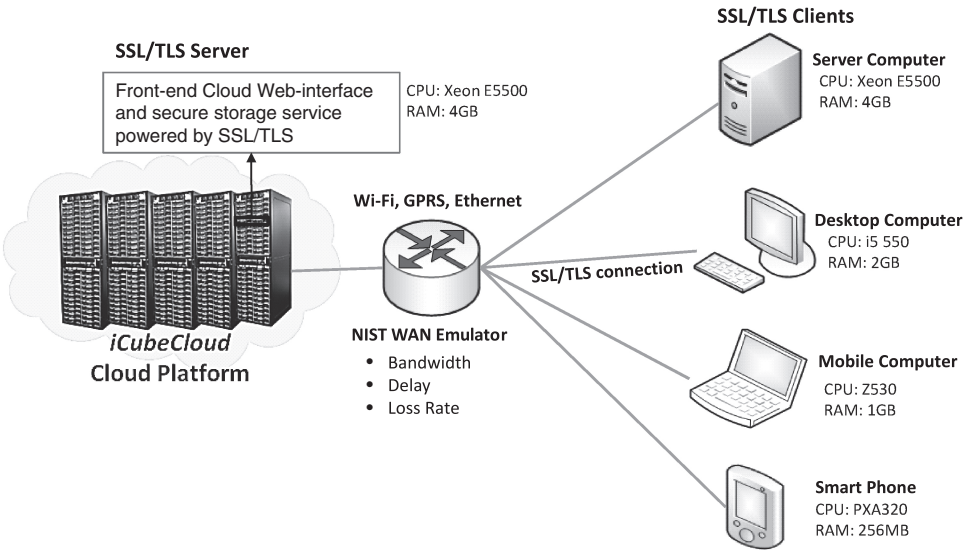

Fig. 9. Experiment environment to measure the performance of ACCENT.

4.1. Various Available Network and Computation Resources of TTSC

The first experiment measures the performance of TTSC as the number of simultaneous client connections is varied. Increases in the simultaneous connections change the utilization of the computation and communication resources in the server and clients. That behavior enables us to observe how TTSC operates with respect to changes in the computation and network load. When an experiment is initiated, all clients establish SSL/TLS connections and then download the target chunk of data (731MB) as rapidly as possible. Figures 10(a) and 10(b) show the overall CPU and network utilization of the server side, respectively, as the number of simultaneous client connections increases. Both parameters increase until the concurrent connection is saturated by the given network or computation resources. In the case of the SSL without compression, the CPU utilization slows down from 50% to 44%, and the network utilization slows down from 72% to 50% when the simultaneous connection is 120 and 280, respectively. In the case of the SSL connections with compression, the CPU utilization slows down from 72% to 44%, and the network utilization slows down from 23% to 10% when the simultaneous connection is 40 and 280, respectively. Our observation of the saturation points confirms that the compression-disabled SSL/TLS may saturate the network bandwidth while leaving the CPU underutilized and vice versa. Specifically, 50% of the CPU time is idle when there are 120 connections, and the CPU idle time increases as the number of simultaneous connections increases to 280.

The goal of TTSC is to take advantage of these underutilized resources. In the case of TTSC-enhanced SSL/TLS, even when the network bandwidth is saturated, the CPU utilization is higher than the two other schemes. The CPU and network utilization are increased as the number of simultaneous connections is increased by up to 80. Figure 10(a) confirms that as the number of simultaneous connections increases, the TTSC improves the CPU utilization by 44.43% more than the SSL/TLS with compression. Figure 10(b) confirms, from the network utilization perspective, that TTSC improves the network utilization by up to 2.91 times the corresponding improvement with the SSL/TLS with compression.

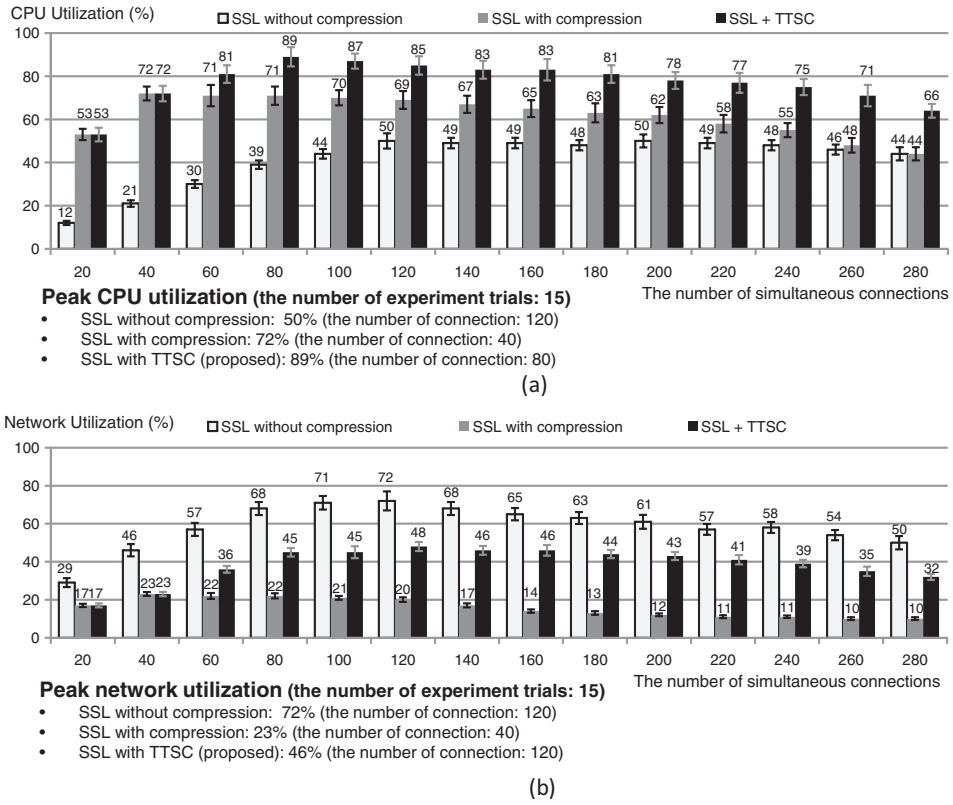

Fig. 10. Resource utilization of the server side with varying the number of concurrent connections. (a) CPU utilization and (b) Network utilization.

In all three cases of Figure 10, the SSL/TLS connections cannot completely saturate the network and the CPU utilization. The cause of the underutilization of the network and the CPU resources is the increasing I/O and context switching overhead as the number of SSL/TLS connections increases. This reflects the operating system involvement in the SSL/TLS-based data delivery process [Coarfa et al. 2006].

Figure 11 compares the effective throughput of the three variants as the number of simultaneous client connections from the server side is increased. We note that the effective throughput is the throughput considering the compression effect. When the number of concurrent connections exceeds 80 simultaneous connections, the effective throughput values of the three cases are decreased. When the number of concurrent connections is 120, the peak effective throughput of TTSC is 948.2Mbps. The performance improvement by TTSC ranges from 138.5Mbps to 245.9Mbps as compared to the SSL without compression and from 0.24Mbps to 552.6Mbps as compared to the SSL with compression.

4.2. Client-Aware Adaptiveness Evaluation of FSCN

In an attempt to evaluate the adaptiveness of FSCN, we used a variable network bandwidth in connections between a server and clients; the variable computation workloads were used as experimental parameters. This process enabled us to evaluate FSCN under a dynamic network environment with variations in the available network bandwidth or the available computation resources of the server and client side. The available

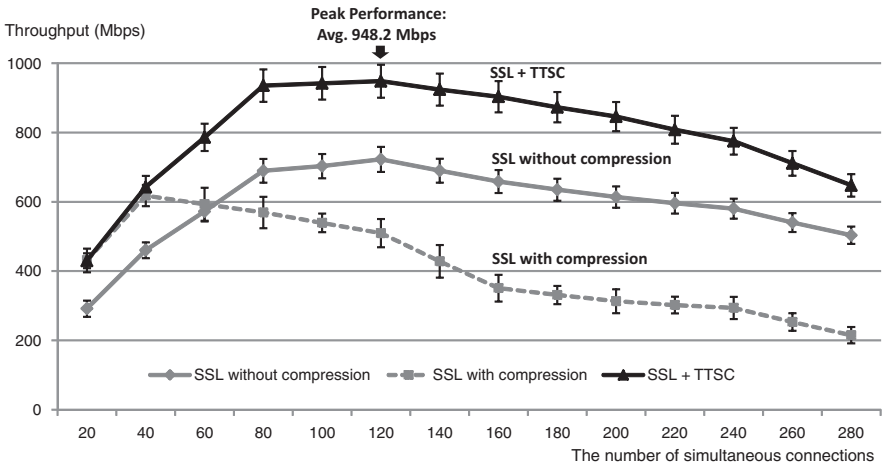

Fig. 11. Effective throughput of server side with varying the number of concurrent connections.

network resources can change regularly: examples include a user who connects a mobile device to the Internet over a gigabit network at the office, a user who takes a mobile device to work and connects via a GPRS network, or a user who makes a Wi-Fi connection while attending a conference. We assume average bandwidths of 250Kbps for the GPRS network, 35Mbps for the Wi-Fi, and 823Mbps for the gigabit Ethernet [NOVARUM 2009]. At the same time, the available computation resources of the server and client side are changed by several factors such as other processes or updates. The changes are shown in Figures 12(a) and 12(b).

To evaluate matching, we compare their performances in terms of the matching ratio between the ideal solution (which was predetermined on the basis of separate trials) and the FSCN solution. Before starting the network and computation pattern, we establish an SSL/TLS connection and set the negotiation duty cycle to 200ms. We then run the server for around 100s to compare the ideal solution with the FSCN solution. Figure 12(c) illustrates the matching ratio of the ideal solution and the FSCN solution. FSCN performs in a similar manner to the best performing technique in all scenarios; it achieves 91.32% probability for the current network bandwidth and computation power even when the network and computation environment are changed dynamically. This observation confirms that FSCN can tolerate network and computation load variations as the most profitable compression module is applied to the corresponding SSL/TLS connection.

4.3. Bandwidth Effect of FSCN

To highlight how FSCN improves the effective bandwidth when the compression module is cognitively applied, we empirically compare the effective FSCN-enhanced transmission rate with the following commonly used SSL/TLS recording schemes: no compression, RLE, lzo, gzip, and bzip. We note that the effective transmission rate is the transmission rate considering the compression effect. Given different levels of network and computation load as the scenario of Figure 12, the FSCN-enhanced transmission rate is evaluated in terms of the previous scenario. When it comes to changing the

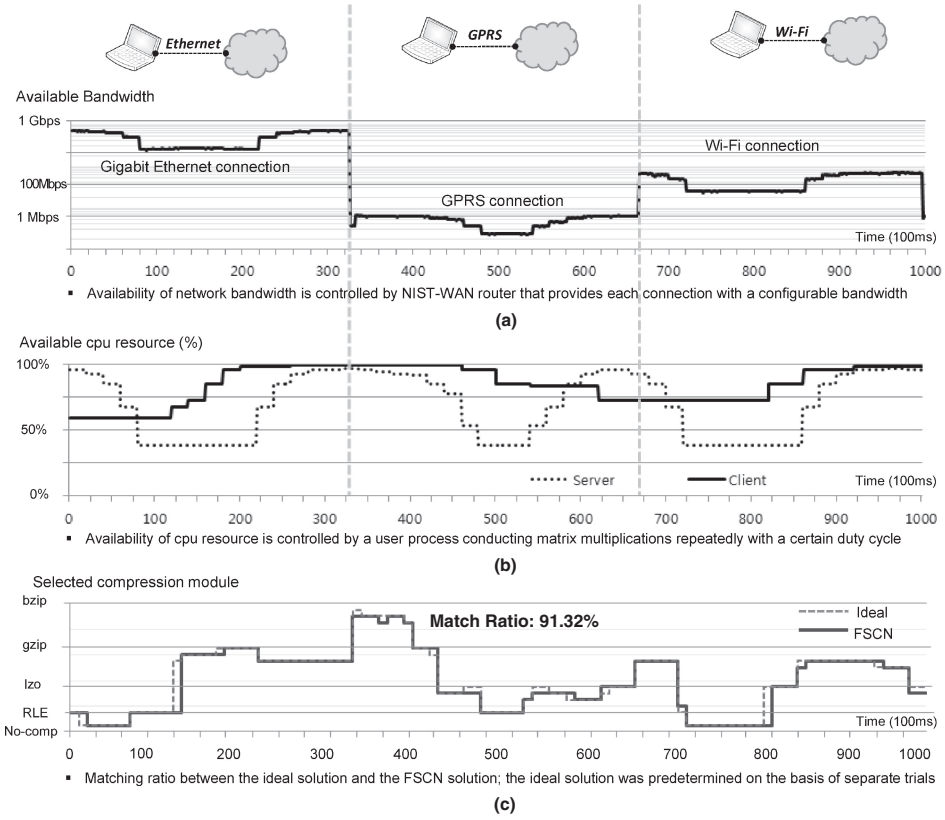

Fig. 12. FSCN adaptivity evaluation. (a) Available bandwidth (b) Available CPU resource (c) Negotiated compression module by FSCN.

negotiation parameter, our rule of thumb is to derive the most profitable compression module that can achieve the highest possible transmission rate.

Figure 13 shows the effective transmission rates for the given scenario. The bars from left-to-right represent the effective transmission rate of commonly used SSL/TLS recording schemes (no compression, RLE, lzo, gzip, and bzip), as well as FSCN, FSCN with TTSC, and the ideal recording scheme. The effective transmission rates in Figure 13 are for the client side; they were measured with four kinds of client machines with diverse hardware specifications, from handheld devices to the server. In the native SSL/TLS recording schemes, the effective transmission rates are from 74.9Mbps to 193.4Mbps. In contrast, FSCN improves the transmission rate from 192.2Mbps to 272.1Mbps. FSCN with TTSC outperforms the other two methods with a transmission rate range of 234Mbps to 358.3Mbps. This range represents a transmission rate that is 40.7% to 156.6% better than that of TTSC only and 85.3% to 212.4% better than the native SSL/TLS recording schemes. Last, in comparison with the ideal scheme (which was predetermined on the basis of a solution derived from the separate trials discussed in the previous section), FSCN achieves a 96.37% transmission rate and FSCN with TTSC achieves a 96.48%.

These observations confirm that we can achieve a higher transmission rate by applying the compression module with FSCN. Furthermore, FSCN can perform in a similar manner to the best-performing technique in the given scenario. The key contribution

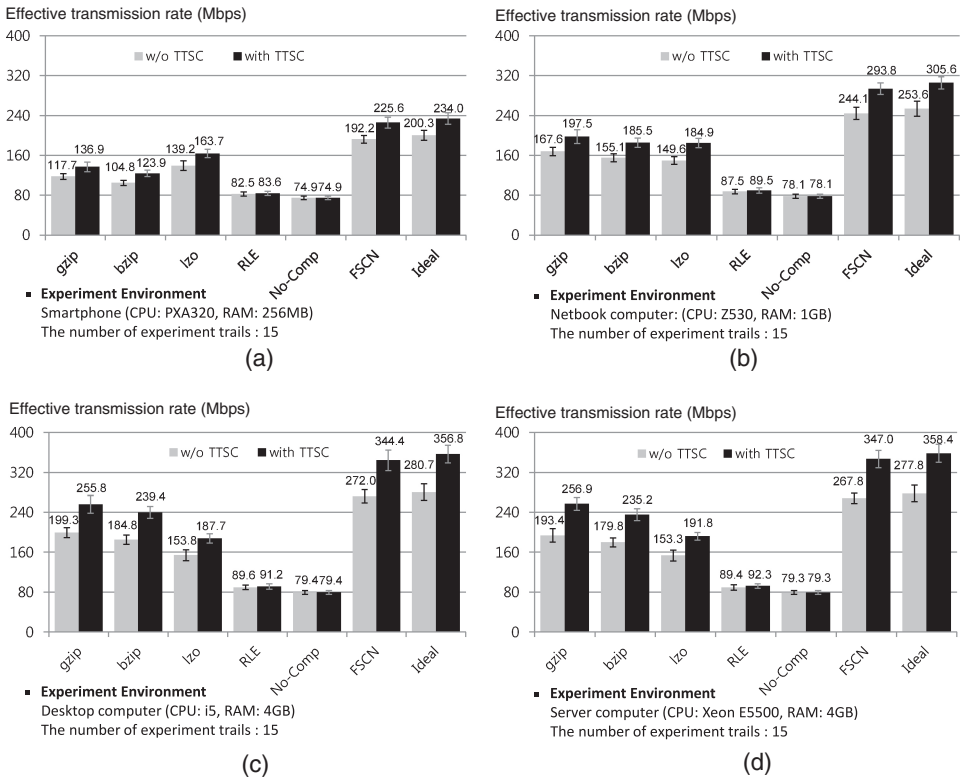

Fig. 13. Transmission rate with FSCN. (a) Smartphone (CPU: PXA320, RAM: 256MB), (b) Netbook computer (CPU: Z530, RAM: 1GB), (c) Desktop computer (CPU: i5, RAM: 4GB), and (d) Server computer (CPU: Xeon 5500, RAM: 4GB).

of FSCN is that it can adapt to wherever the connection and computation workload are different and dynamic.

4.4. Switching Overhead Reduction by UMAC

UMAC enables an SSL/TLS subsystem to switch a compression module with a lower switching overhead by means of a unified memory instance. Figure 14 shows how the overhead is minimized by UMAC in the client side during the FSCN-induced compression switching. At the beginning of this experiment, a client (CPU: Z530, RAM: 1GB) established a connection with a server (CPU: Xeon E5500, RAM: 4GB) and let the connection run for 10s. After 10ms, the network and computation environment is changed to force the switching compression module to change from bzip to lzo. Without UMAC, it takes approximately 7.34ms to reach the maximum throughput of the corresponding compression module. UMAC reduces the time delay by 1.22ms. When a renegotiation message is received, a new compression module is immediately implemented without the memory allocation and release operations.

We measured the compression switching overhead with the four kinds of client machines as shown in Figures 13(b) to 13(d). In the native compression switching, the average switching delay is from 1.48ms to 9.02ms on low computing devices (Figure 14(b) and Figure 14(c)). The range of the switching overhead enhanced with UMAC (0.34ms to 0.49ms) achieves a 4.35 to 18.41 times shorter switching delay than that of native compression switching. Even in the case of high-end computing devices (Figure 14(d)

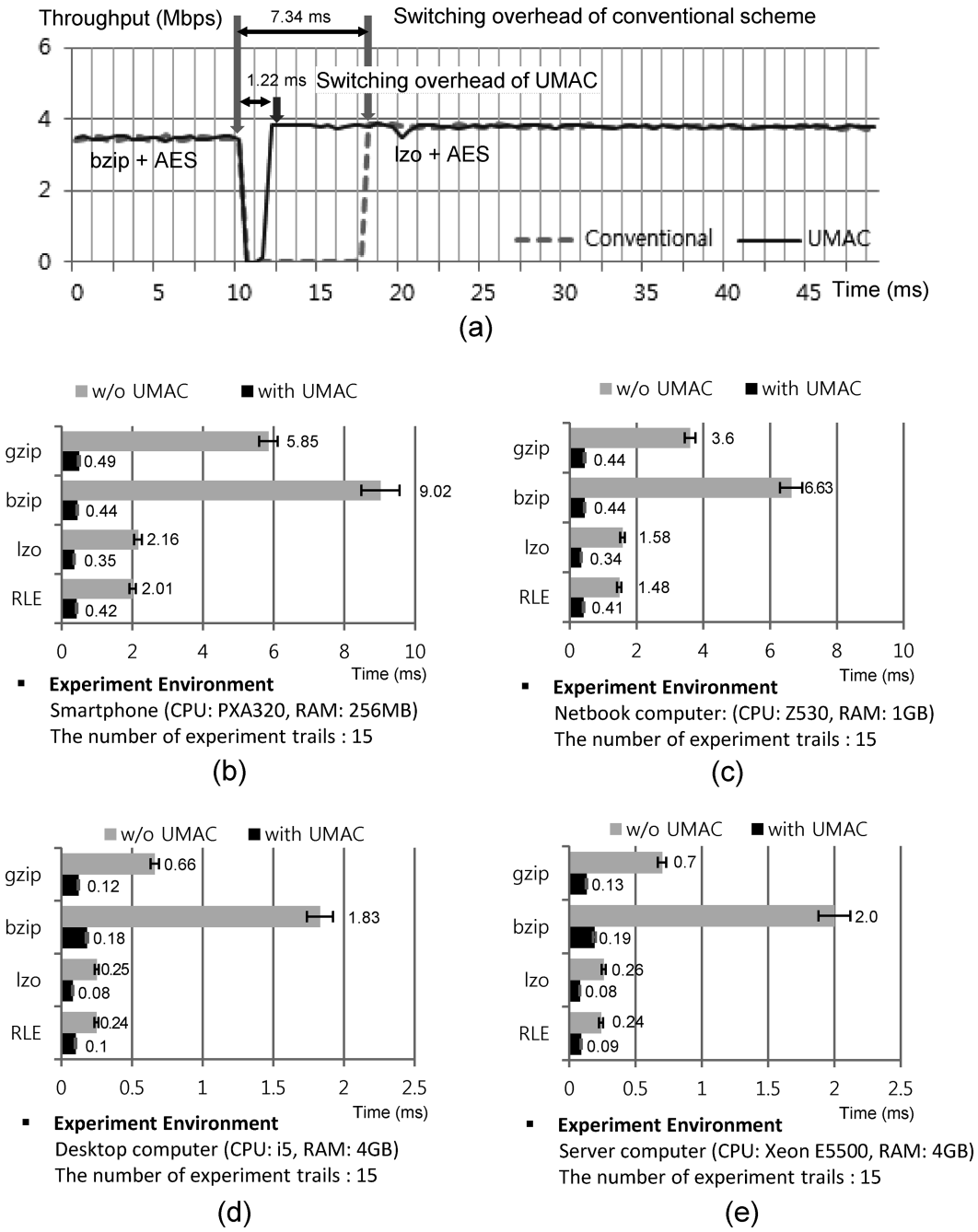

Fig. 14. The performance impact of UMAC. (a) The throughput against switching overhead, (b) Smartphone (CPU: PXA320, RAM: 256MB), (c) Netbook computer, (CPU: Z530, RAM: 1GB), (d) Desktop computer (CPU: i5, RAM: 4GB), (e) Server computer (CPU: Xeon 5500, RAM: 4GB).

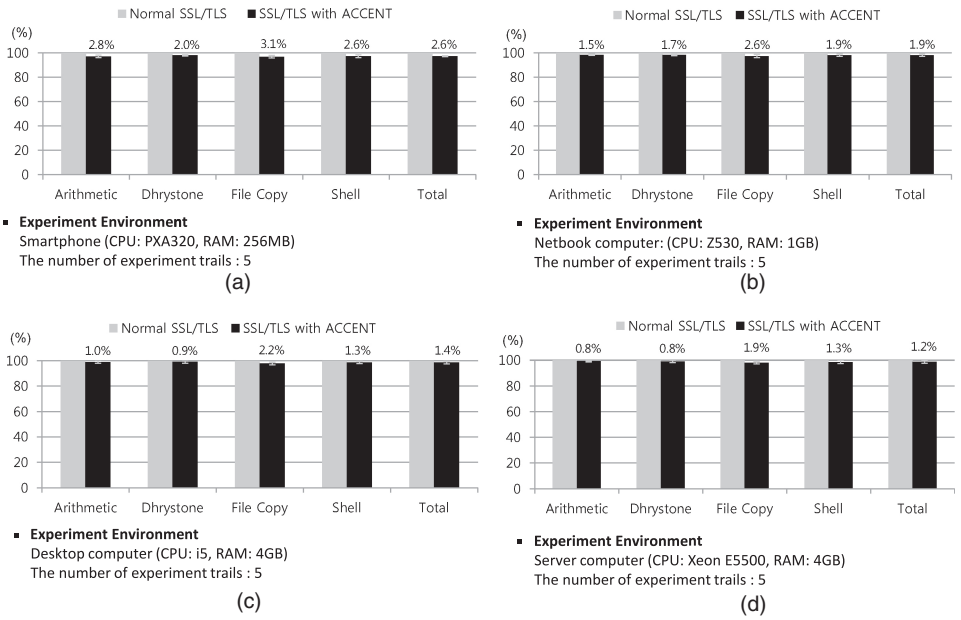

Fig. 15. The normalized performance impact of ACCENT in comparison with the normal SSL/TLS as the baseline. (a) Smartphone (CPU: PXA320, RAM: 256MB), (b) Netbook computer (CPU: Z530, RAM: 1GB), (c) Desktop computer (CPU: i5, RAM: 4GB), (d) Server computer (CPU: Xeon 5500, RAM: 4GB).

and Figure 14(e)), the average switching delay is from 0.24ms to 2ms. The range of the average switching delay enhanced with UMAC is 0.08ms to 0.19ms. As a result, UMAC achieves a 3 to 10.53 times shorter switching delay than the native compression switching scheme.

Our results confirm that UMAC can minimize the compression switching overhead. In addition, our profiling shows that during compression switching the memory allocation and release operations are removed from the native compression switching. Clearly, this performance result confirms that UMAC enables FSCN to operate more seamlessly in the face of abrupt changes in an SSL/TLS connection.

4.5. Performance Impact of ACCENT

The goal of our next experiment is to examine the performance impact when ACCENT is applied to the SSL/TLS protocol stack. The main purpose is to calculate the performance overhead that is imposed by ACCENT. In this experiment, we repeatedly run the network and computation load variation as in the scenario of Figure 12. The performance impact of ACCENT is measured using the UnixBench [Jon Tombs et al. 2011] benchmark. We measure the total execution time of the benchmark when ACCENT is applied to the SSL/TLS protocol stack and compare it to the total execution time of the benchmark when the normal SSL/TLS protocol is applied. We note that the normal SSL/TLS protocol is scheduled to perform compression switching as predetermined on the basis of the FSCN solution. It is to measure the performance overhead imposed by ACCENT apart from the compression overhead. The results of our experiment are shown in Figure 15. The black bars represent the normalized execution time when ACCENT is applied to the SSL/TLS protocol stack in comparison with the normal SSL/TLS scheme as the baseline (100%). The measured execution times in Figure 15 were measured with four kinds of client machines with diverse hardware specifications, from handheld devices to the server. The performance result implies from 0.8%

to 3.1% performance overhead if ACCENT is applied to the SSL/TLS protocol stack. This can be attributed to the very low overhead imposed by ACCENT because as it can achieve an improvement in the transmission rate of 85.3% to 212.4% (as described in Section 4.3) with less than 3.1% system overhead.

5. CONCLUSION

Our aim was to provide a full-fledged cognitive SSL/TLS with heterogeneous environments (device, network, and cloud) and workloads awareness for a cloud computing environment. To accomplish this task, we have thoroughly analyzed the SSL/TLS-based data communication in an SSL/TLS protocol stack, which is a standard protocol for secure Internet communication. The SSL/TLS-based system has three rudimentary operations. Our ACCENT system is an improvement on these rudimentary operations. It has three advanced mechanisms: an efficient SSL/TLS transmission routine, an SSL/TLS dynamics-aware data compression mechanism, and a unified memory allocation for frequent compression switching. In order to ensure backward compatibility, we patched the internal codes of the SSL/TLS core routines so that existing SSL/TLS-based applications require no modification.

Thus, our implementation in OpenSSL is feasible and practical. In addition, the experimental results show the improved performance in terms of throughput, resource utilization, and adaptiveness to dynamic changes in SSL/TLS connections. Three mechanisms can be selectively adopted in accordance with the characteristics of target applications to improve a secure network system. For instance, for a static network environment, TTSC mainly helps to improve the performance of secure data transfer because the other two mechanisms, FSCN and UMAC, are designed for a dynamic network environment. In contrast, a Web-based storage service and content delivery service, which concurrently support where heterogeneous computing devices and networks are connected to each other, can improve their overall performance by utilizing FSCN and UMAC. We believe that ACCENT can facilitate the cloud-based services with an adaptivity, such as a Web-based storage service and a content delivery service over SSL/TLS as well as emerging applications, such as cloud-based mobile office and cloud storage. This type of facilitation is possible as long as the relevant applications utilize SSL/TLS as an underlying security layer.

ACKNOWLEDGMENTS

The authors wish to thank their anonymous referees for all of their invaluable comments and suggestions. They would also like to NexR Co. Ltd for their help in conducting some of the experiments in this study.

REFERENCES

ALAIDAROS, H., RASID, M., OTHMAN, M., AND ABDULLAH, R. 2007. Enhancing security performance with parallel crypto operations in ssl bulk data transfer phase. In *Proceedings of the Telecommunications and Malaysia International Conference on Communications (ICT-MICC'07)*. IEEE, 129–133.

AMAZON-EC2. 2011. Amazon elastic compute cloud ec2. http://aws.amazon.com/ec2/.

AMAZON-S3. 2011. Amazon simple storage service s3. http://aws.amazon.com/s3/.

BERBECARU, D. 2005. On measuring ssl-based secure data transfer with handheld devices. In *Proceedings of the 2nd International Symposium on Wireless Communication Systems*. 409–413.

BRIK, V., MISHRA, A., AND BANERJEE, S. 2005. Eliminating handoff latencies in 802.11 wlans using multiple radios: applications, experience, and evaluation. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement (IMC'05)*. USENIX Association, Berkeley, CA, 27–27.

CARSON, M. AND SANTAY, D. 2003. Nistnet: a linux-based network emulation tool. *SIGCOMM Comput. Comm. Rev. 33*, 111–126.

CASTELLUCCIA, C., MYKLETUN, E., AND TSUDIK, G. 2006. Improving secure server performance by re-balancing ssl/tls handshakes. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'06)*. ACM, New York, NY, 26–34.

CHOU, W. 2002. Inside ssl: accelerating secure transactions. *IT Professional 4,* 5, 37–41.

COARFA, C., DRUSCHEL, P., AND WALLACH, D. S. 2006. Performance analysis of TLS web servers. *ACM Trans. Comput. Syst. 24*, 39–69.

DE BRUIJN, W. AND BOS, H. 2008. Pipesfs: fast linux i/o in the unix tradition. *SIGOPS Oper. Syst. Rev. 42*, 55–63.

DROPBOX. 2011. Dropbox: Online backup, file sync, and data sharing. http://www.dropbox.com.

GAILLY, J.-L. 2010. Gzip file format specification and compression library Ver. 1.4. http://www.gnu.org/software/gzip/.

GEELNARD, M. 2006. Basic compression library (bcl) Ver. 1.2.0. http://bcl.comli.eu/home-en.html.

GILCHRIST, J. 2009. Archive comparison test (act) Ver. 2.0. http://www.compression.ca/act/.

GOOGLEDOCS. 2011. Google docs: Online documents, spreadsheets, presentations. http://docs.google.com.

HALEEM, M. A., SUBBALAKSHMI, K. P., AND CHANDRAMOULI, R. 2007. Joint encryption and compression of correlated sources with side information. *EURASIP J. Inf. Secur. 11,* 1–11:9.

JANG, K., HAN, S., HAN, S., MOON, S., AND PARK, K. 2011. Sslshader: cheap SSL acceleration with commodity processors. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11).* USENIX Association, Berkeley, CA, 1–1.

JEANNOT, E. AND KNUTSSON, B. 2002. Adaptive online data compression. In *Proceedings of the 11th International Symposium on High Performance Distributed Computing (HPDC-11'02)*. IEEE, 379–388.

JOHN DUNLOP, R. E. AND ABANI, P. 2010. Cloud computing: How client devices affect the user experience. Tech. rep., Intel. http://communities.intel.com/docs/DOC-5672.

JON TOMBS, BEN SMITH, R. G. AND YAGER, T. 2011. Unixbench: Unix benchmark suite (arithmetic, dhrystone, storage, shell). http://code.google.com/p/byte-unixbench/.

KANT, K., IYER, R., AND MOHAPATRA, P. 2000. Architectural impact of secure socket layer on internet servers. In *Proceedings of the International Conference on Computer Design*. 7–14.

KOUNAVIS, M. E., KANG, X., GREWAL, K., ESZENYI, M., GUERON, S., AND DURHAM, D. 2010. Encrypting the internet. *SIGCOMM Comput. Comm. Rev. 40*, 135–146.

KRINTZ, C. AND SUCU, S. 2006. Adaptive on-the-fly compression. *IEEE Trans. Paral. Distrib. Syst. 17,* 1, 15–24.

LEE, H. K., MALKIN, T., AND NAHUM, E. 2007. Cryptographic strength of ssl/tls servers: current and recent practices. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC'07)*. ACM, New York, NY, 83–92.

LEVON, J. AND ELIE, P. 2011. Oprofile: A system profiler for linux. http://oprofile.sourceforge.net/.

MICROSOFT-AZURE. 2010. Microsoft azure cloud platform. http://www.microsoft.com/windowsazure/.

MORALES-SANDOVAL, M. AND FEREGRINO-URIBE, C. 2005. A hardware architecture for elliptic curve cryptography and lossless data compression. In *Proceedings of the 15th International Conference on Electronics, Communications and Computers (CONIELECOMP'05).* 113–118.

NETCRAFT. 2010. Netcraft web server survry.

NEXR-ICUBECLOUD. 2011. icubecloud: Cloud computing and elastic-storage services. https://testbed.icubecloud.com/.

NOVARUM. 2009. 3g speed and reliability results by city. http://www.pcworld.com/.

OBERHUMER, M. F. 2010. Lzo 2.04 compression library. http://www.oberhumer.com/opensource/lzo/.

OKAMOTO, N., KIMURA, S., AND EBIHARA, Y. 2003. An introduction of compression algorithms into ssl/tls and proposal of compression algorithms specialized for application. In *Proceedings of the 17th International Conference on Advanced Information Networking and Applications (AINA'03).* 817–820.

OPENSSL.ORG. 2011. Openssl: The open source toolkit for ssl/tls Ver. 1.0.0. http://www.openssl.org/.

PARK, K.-W., PARK, S. K., HAN, J., AND PARK, K. H. 2010. Themis: Towards mutually verifiable billing transactions in the cloud computing environment. In *Proceedings of the IEEE 3rd International Conference on Cloud Computing (CLOUD'10)*. IEEE Computer Society, Los Alamitos, CA, 139–147.

QI, F., TANG, Z., WANG, G., AND WU, J. 2009. Qos-aware optimization strategy for security ranking in ssl protocol. In *Proceedings of the IEEE 6th International Conference on Mobile Adhoc and Sensor Systems (MASS'09.)* 842–847.

RESCORLA, E. 2001. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

SEWARD, J. 2010. bzip compression library Ver. 1.0.6. http://bzip.org/.

SHACHAM, H., BONEH, D., AND RESCORLA, E. 2004. Client-side caching for tls. *ACM Trans. Inf. Syst. Secur. 7*, 553–575.

SPEC. 2010. Specweb2009: Spec benchmark for evaluating web server performance. http://www.spec.org/web2009/.

W3TECHS. 2011. Usage of compression for websites. http://w3techs.com/.

WISEMAN, Y., SCHWAN, K., AND WIDENER, P. 2005. Efficient end to end data exchange using configurable compression. *SIGOPS Oper. Syst. Rev. 39*, 4–23.

WU, C.-P. AND KUO, C.-C. 2005. Design of integrated multimedia compression and encryption systems. *IEEE Trans. Multimedia 7,* 5, 828–839.

WU, L., WEAVER, C., AND AUSTIN, T. 2001. Cryptomaniac: a fast flexible architecture for secure communication. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. ACM, New York, NY, 110–119.

ZHAO, L., IYER, R., MAKINENI, S., AND BHUYAN, L. 2005. Anatomy and performance of ssl processing. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05).* 197–206.

Received April 2011; revised September 2011; accepted October 2011