Hindawi
Security and Communication Networks
Volume 2018, Article ID 7631342, 24 pages
https://doi.org/10.1155/2018/7631342

Research Article

Hardware/Software Adaptive Cryptographic Acceleration for Big Data Processing

Chunhua Xiao[1,2] **Lei Zhang**[1] **Yuhua Xie**[1] **Weichen Liu**[1,2] **and Duo Liu**[1,2]

[1]*Department of Computer Science, Chongqing University, Chongqing 400044, China*
[2]*Key Laboratory of Dependable Service Computing in Cyber Physical Society of Ministry of Education, Chongqing 400044, China*

Correspondence should be addressed to Chunhua Xiao; xiaochunhua@cqu.edu.cn

Received 9 February 2018; Revised 23 July 2018; Accepted 6 August 2018; Published 27 August 2018

Academic Editor: Jun Zhou

Along with the explosive growth of network data, security is becoming increasingly important for web transactions. The SSL/TLS protocol has been widely adopted as one of the effective solutions for sensitive access. Although OpenSSL could provide a freely available implementation of the SSL/TLS protocol, the crypto functions, such as symmetric key ciphers, are extremely compute-intensive operations. These expensive computations through software implementations may not be able to compete with the increasing need for speed and secure connection. Although there are lots of excellent works with the objective of SSL/TLS hardware acceleration, they focus on the dedicated hardware design of accelerators. Hardly of them presented how to utilize them efficiently. Actually, for some application scenarios, the performance improvement may not be comparable with AES-NI, due to the induced invocation cost for hardware engines. Therefore, we proposed the research to take full advantages of both accelerators and CPUs for security HTTP accesses in big data. We not only proposed optimal strategies such as data aggregation to advance the contribution with hardware crypto engines, but also presented an Adaptive Crypto System based on Accelerators (ACSA) with software and hardware codesign. ACSA is able to adopt crypto mode adaptively and dynamically according to the request character and system load. Through the establishment of 40 Gbps networking on TAISHAN Web Server, we evaluated the system performance in real applications with a high workload. For the encryption algorithm 3DES, which is not supported in AES-NI, we could get about 12 times acceleration with accelerators. For typical encryption AES supported by instruction acceleration, we could get 52.39% bandwidth improvement compared with only hardware encryption and 20.07% improvement compared with AES-NI. Furthermore, the user could adjust the trade-off between CPU occupation and encryption performance through MM strategy, to free CPUs according to the working requirements.

1. Introduction

As we enter the big data era, network data demonstrates an explosive growth [1, 2]. More and more transactions, such as e-commerce and net-banking, require the transfer of sensitive information via the Internet, and the security is becoming more and more important for web applications [3, 4]. The Transport Layer Security (TLS) protocol and its successor Transport Layer Security (TLS) can be used to secure applications that communicate over a network and are currently predominant protocols for sensitive accesses. The most widely deployed, freely available implementation of the SSL/TLS protocol is the OpenSSL library [5]. The core library of OpenSSL implements basic cryptographic functions and provides various utility functions. However, its crypto functions, such as symmetric key ciphers and hash algorithms, are extremely compute-intensive operations [6, 7]. OpenSSL does these expensive computations through software implementations and it may not be able to compete with the increasing need for speed and secure connections for web services [8]. Among the many solutions pursued to overcome this problem, one is to apply hardware accelerators to perform the cryptographic (crypto) operations. Implemented in hardware, these crypto accelerators are tamper-proof and difficult to clone, thus providing added security bonus [9].

In literature, there are lots of works published with the objective of SSL/TLS hardware acceleration [10–16]. Some of them focus on the dedicated hardware design for accelerators.

For example, work in [10] implemented AES acceleration for embedded systems. Work in [11] presented a crypto hash SHA-2 logic core with reconfigurable hardware. Research in [12] accelerated elliptic curve cryptography through a very cheap FPGA. In [13], the cipher functions used in the SSL-driven connection, including Scalable Encryption Algorithm (SEA), Message Digest Algorithm (MD5), and Secured Hash Algorithm (SHA2) are accelerated in the VLSI Cryptosystem through FPGA. Others mount all processes for SSL/TLS ciphered communication into a single FPGA or ASIC, such as works in [14–16]. These works showed a great performance improvement compared with software encryption with crypto lib. Nonetheless, these studies concentrated on the implementation of the hardware itself and hardly referred to how to utilize crypto accelerators efficiently with least cost, not to talk about the design methodology for taking full advantages of both accelerators and CPUs. Furthermore, most hardware accelerations for SSL/TLS are specified for embedded systems [10–13], which could not satisfy the high volume and high concurrent accesses requirements in big data age.

Although a lot of work has been done for hardware acceleration, how to utilize them efficiently in the new application scenario of big data is still an open problem. Besides, with AES-NI (Advanced Encryption Standard Instruction Set, or the Intel Advanced Encryption Standard New Instructions), the software crypto computations could also be accelerated, which could provide comparable performance with accelerators for web applications for small data requirements [17]. Therefore, we proposed the research to take full advantages of both accelerators and CPUs to ensure the security HTTP access in big data. Firstly, we are in need of implementation of a prototyping system with multiple CPUs and encryption accelerators, on the basis of which, software and hardware computing overhead are analyzed in detail. Then, we proposed interrupt aggregation and data aggregation to reduce the working cost of hardware invocation. Finally, we put forward adaptive scheduling and MM to optimize the utilization of both software and hardware crypto engines. The main contributions of this work are as follows:

(i) To the best of our knowledge, this is the first Adaptive Crypto System based on Accelerators (ACSA) with software and hardware codesign, which is able to adopt crypto mode adaptively and dynamically according to the request character and system load. The problems of interrupt optimization, data aggregation, and adaptive scheduling for rational resource allocation are also carefully considered.

(ii) We improved the kernel for ACSA to solve the additional overhead caused by hardware acceleration. A dynamic interrupt aggregation is added to the kernel to reduce the system cost induced by high frequency of hardware interrupts.

(iii) We proposed a resource allocation strategy with MM algorithm (maximize utilization with minimal overhead) and adaptive scheduling, to maximum overall encryption bandwidths. The designer/user can find the most reasonable process scheduling strategy through MM for heterogeneous encryption platforms, while adaptive scheduling is applicable to play their respective advantages of hardware/software encryption.

(iv) Through the establishment of 40Gbps networking, we are able to evaluate the system performance in real applications with the high workload on various benchmarks and system configurations. For the encryption algorithm 3DES, which is not supported in AES-NI, we could get about 12 times acceleration with accelerators. For typical encryption AES supported by instruction acceleration, we could get 52.39% bandwidth improvement compared with only hardware encryption, and 20.07% improvement compared with AES-NI. Furthermore, the user could adjust the trade-off between CPU occupation and encryption performance through MM strategy, to free CPUs according to the working requirements.

(v) Proposed design methodology possesses universal properties to some extent. The design flow of ACSA and MM algorithm is applicable to other similar designs with hardware acceleration for secure web access. As long as the design is heterogeneous architecture with CPUs and accelerators, the proposed design methodology is applicable regardless of the type of CPU and accelerator. Besides, this work is based on the ARM server, which can be furthered for energy efficiency exploration, providing emerging solutions for data center besides X86 based architectures.

The remainder of this paper is organized as follows: Section 2 discusses the related works. Section 3 shows the system architecture and design methodology of ACSA, in which Crypto algorithm adaption and interrupt aggregation are also introduced in this section. Sections 4 and 5 discuss the methodology we proposed for adaptive scheduling and optimal resource allocation. We present the evaluation means and experimental results in Section 6. Finally, we conclude our discussions in Section 7.

2. Related Work

Most effective works have been proposed for SSL/TLS processes acceleration. We distinguish them into three categories according to the hardware implementation.

The first one focuses on the acceleration for a specific algorithm in SSL/TLS process. Notable among them are [10–12]. In work [10], Praveen Kumar B. et.al designed and verified the working AES crypto Verilog core, which runs on top of an embedded Linux distribution, uClinux. The OpenSSL library has been ported and cross-compiled to work with only AES that processes data blocks of 128 bits using a cipher key of length 128, 192, or 256 bits. Work in [11] presented the implementation of a crypto hash SHA-2 logic core in reconfigurable hardware, and a throughput of 644 Mbits/sec could be achieved by this design. In another work [12], the researchers demonstrated a reconfigurable hardware accelerator for OpenSSL's implementation of ECC

and showed how a low-cost hardware platform is sufficient to double performance.

The second kind of acceleration researches further integrates typical cipher algorithms in a single microchip. For example, Mohamed Khalil-Hani in work [18] integrates the AES-256, SHA-1, SHA-2, RNG, and RSA-2048 cryptographic hardware cores into one FPGA microchip for an embedded system. The work proposed by [19] developed a hardware prototype of the cryptographic processor in FPGA technology. In [13], the cipher functions used in SSL-driven connection, including Scalable Encryption Algorithm (SEA), Message Digest Algorithm (MD5), and Secure Hash Algorithm (SHA2), are accelerated in the VLSI Cryptosystem through FPGA.

The third type mounts all processes for SSL/TLS ciphered communication into one ASIC, which is usually denoted as Network Security Processors (NSPs). NSP performs various cryptographic operations specified by network security protocols and helps to offload the computation intensive burdens from Network Processors (NPs). Literature [14] presents a security processor to accelerate cryptographic processing in modern security applications, which is capable of popular cryptographic functions such as RSA, AES, hashing, and random number generation. Research in [15] shows the 10Gbps implementation of low-power SSL/TLS accelerator on 65 nm FPGA. The usage of FPGA/ASIC enables highly efficient processing and low-power consumption by using parallel optimization and pipelined processing. Work in [16] proposes a high-performance NSP Zodiac intended for both IPsec and SSL protocols acceleration, which is synthesized with a 0.18μm CMOS technology.

There is no doubt that these works make effective efforts for the acceleration in security HTTP accesses. However, they concentrated more on the hardware implementation itself and hardly referred to how to utilize crypto accelerators efficiently with the least cost. To the best of our knowledge, we did not find any clear illustration for the performance comparison between hardware accelerators and AES-NI, not to talk about the design methodology for taking full advantages of both accelerators and CPUs. Furthermore, most works, especially types 1 and 2 mentioned above focus on the embedded applications, which could not satisfy the high volume and high concurrent accesses requirements in big data age. Although type 3 could achieve higher throughput with Gigabit per second through application specific design, it is lack in generality and flexibility compared with other solutions. Moreover, it takes extensive design efforts for ASIC or prototype board.

Therefore, we surveyed and analyzed different working flows with SSL/TLS firstly and make the working mode and performance differences clear between hardware accelerators and AES-NI. Based on the exploration of existing works, we proposed Adaptive Crypto System based on Accelerators (ACSA) with software and hardware codesign, which is able to adopt crypto mode adaptively and dynamically according to the request character and system load. The problems of interrupt optimization, data aggregation, and adaptive scheduling for rational resource allocation are also carefully considered. Through the evaluation with the high workload on various benchmarks and system configurations, we could get overall performance improvement compared with only AES-NI or hardware accelerator solutions. Furthermore, the proposed design methodology is applicable to other similar designs to perform a high resource efficiency system for secure HTTP access.

3. Adaptive Crypto System with Accelerators

In the big data age, Web Server applications exhibit a remarkable characteristic of high concurrency and mass data. To accelerate the encryption/decryption process for these applications, we proposed Adaptive Crypto System based on Accelerators for secure HTTPS access. For clarity, we denoted this system as ACSA. ACSA integrates multiple processing cores and crypto accelerators. The processing cores are ARMs in this research. The reason for using the ARM core is for further exploration such as energy efficiency besides X86 architecture [20–22]. Although the server architecture is different, the proposed design methodology is practical to other similar acceleration systems.

Different from traditional cryptosystems, the encryption process in ACSA is adjustable dynamically between hardware engines and CPUs. An Adaptive Scheduler is responsible for working mode adoption. This scheduler chooses reasonable encryption way according to the request character and system resources. The data request aggregation and the MM strategy are considered carefully in our design to make full utilization of both software and hardware computing engines. In order to ensure the versatility, the Adaptive Scheduler also allows the user to choose the working mode flexibly.

We also considered the quality of service. We implemented a fault detection module to guarantee the reliability of hardware crypto. This detection module works if hardware crypto computing is adopted. If the hardware encryption engine breaks down, which could not provide the crypto function properly, the hardware detection module will send a fault signal to the mode-switching module. At this point, the encryption tasks in progress will be interrupted, and working mode changes seamlessly from the hardware encryption to the software computing. These interrupted jobs will be reexecuted through the software encryption units, thereby ensuring the clients can obtain the requested data correctly.

To make full utilization of hardware accelerators, ACSA works at half-sync/half-async mode [23]. As shown in Figure 1, the whole system divides into a synchronous working level (the upper part) and an asynchronous level (the lower part). The two parts exchange information through a synchronous/asynchronous communication layer. In the upper part, Web Server accepts requirements from different clients, generates multiple crypto tasks simultaneously, and then sends these tasks to the synchronous/asynchronous communication layer. Through this synchronous process, ACSA calls SSL/TLS library to establish secure communications. In the lower part, the hardware driver receives crypto tasks from interaction layer and sends them to the hardware engines for encryption. If the crypto computing finished, hardware engine notifies the driver through an interrupt and then sends back cryptograph to sync/async layer via callback functions, so as to awaken the waiting process.

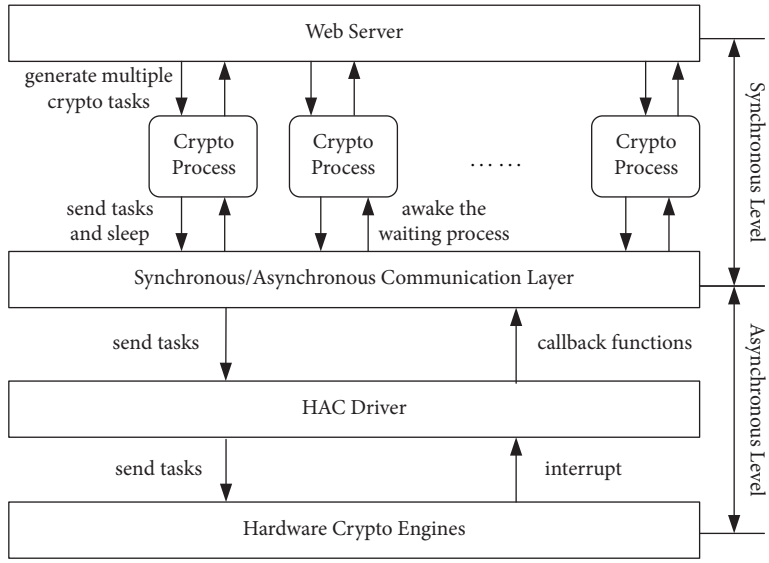


Figure 1: Logical architecture of ACSA.

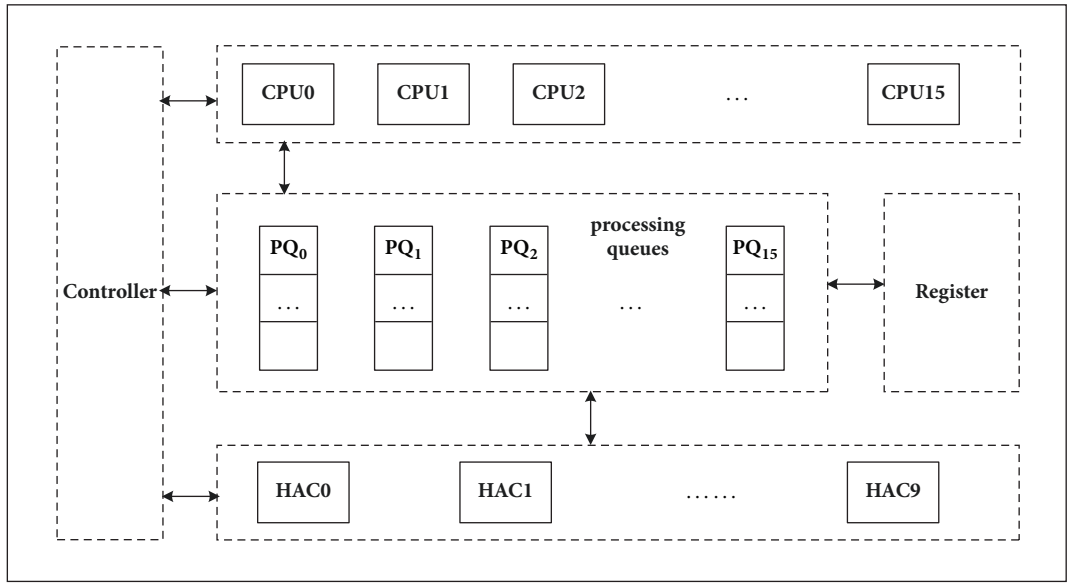


Figure 2: System architecture of ACSA.

3.1. System Architecture of ACSA. We choose a lightweight Web Server TAISHAN as our hardware platform. We enabled 16 ARMs and 10 HACs (Hardware Acceleration Components) for our research. ARMs are Cortex series CPUs. HACs are configured as security accelerators in our ACSA, which support typical encryption algorithms, such as AES-CBC [24], AES-GCM [25], 3DES [26], and SHA1 [26, 27]. 16 processing queues (PQ) are responsible for software and hardware interaction, and each queue is possible to receive multiple requests issued by the engine driver. The software (hardware driver) pushes tasks to crypto engines through PQ to implement required acceleration tasks. Each PQ can be independently enabled, and its starting address and length support individual configuration through registers. The designer/user is able to configure the correspondence between the PQ and CPU flexibly. As shown in Figure 2, CPU utilizes a write pointer to inform the pending tasks in the processing queue. The controller gets an entry from processing queue according to the results of WRR (Weighted Round Robin) arbitration, then resolves tasks, and invokes HACs for execution. Once the computing finished, an interrupt will generate to inform the upper layer for results return.

3.2. Workflow of ACSA. As shown in Figure 3, the working flow of ACSA divides into 5 logic levels: application layer for HTTPS access, OpenSSL layer for SSL/TLS processing, CryptoDev layer, HAC driver, and hardware layer with accelerators, among which, the application layer and the

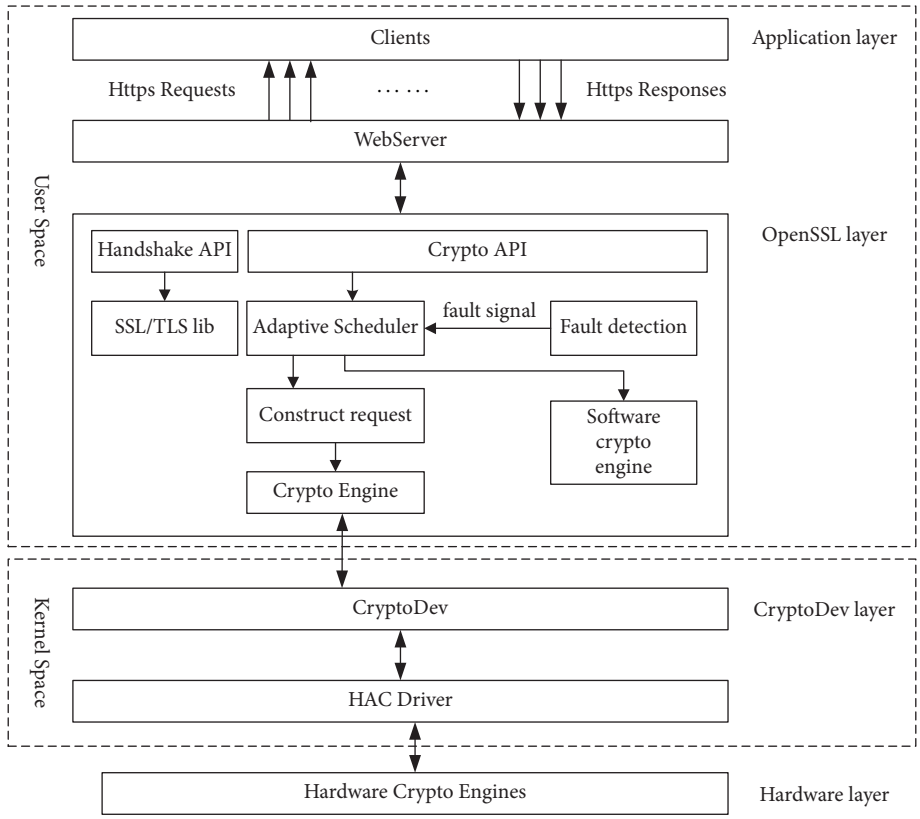


Figure 3: Working flow of ACSA.

OpenSSL layer work in the user space, while the CryptoDev layer and the HAC driver work in the kernel space.

For the application layer, we adopted the typical Web Server, Nginx, for HTTPS service. Web Server monitors client requirements, responds to clients' requests, and is responsible for load balance with multithread. During this process, Web Server authenticates clients' identity and checks for the security through the crypto function provided by OpenSSL [28]. If HTTPS connection is enabled, Web Server will transfer required data to the lower logic layer for further processing.

OpenSSL subsystem responds the crypto requirements from Web Server and interacts with CryptoDev for HACs invocation. The roles of OpenSSL layer in ACSA include the following:

(1) Utilize provided toolkit for secure connection with the SSL/TLS protocols.

(2) Integrate the Adaptive Scheduler for encryption mode switching, which enables the reasonable task scheduling between software computing and hardware encryption.

(3) Extend the feature of resource allocation strategy, MM.

(4) If software computing is adopted, invoke the cryptography library to perform encryption requirements.

(5) Interact with the lower layer CryptoDev. If the required data need to do hardware encryption, encapsulate the client requirements as EVP, and then transmit EVP blocks to CryptoDev through CryptoAPI. After hardware becomes complete, return the generated cryptograph back to the application layer.

CryptoDev subsystem exists as a module in Linux kernel. Upwards, CryptoDev interacts with OpenSSL layer, receives delivered require data, and returns completed cryptograph. Downwards, CryptoDev transforms received EVPs as the data structure that can be identified by the HAC driver. CryptoDev acted as the middle layer between synchronous and asynchronous communications.

Memory copy consumes lots of system resource in hardware acceleration systems, especially mass data interaction occurred between hardware and software. To solve this problem, we used an efficient transmission scheme "zero-copy" in CryptoDev layer. We extended CryptoDev subsystem to support both AEAD (Authenticated Encryption with Associated Data) and non-AEAD encryption requirements. For all requirements, original data and encrypted results are organized as scatterlist. The scatter virtual addresses of DMA buffer are arranged in a list in our design. In this way, data transmission between memory and HACs could be achieved through one DMA operation.

HAC driver acts as a loadable module in Linux kernel and plays an important role in accelerator invocation. The driver performed four major works: initialization of hardware engines, device loading and unloading, encryption algorithms registration, and interrupt processing. The driver

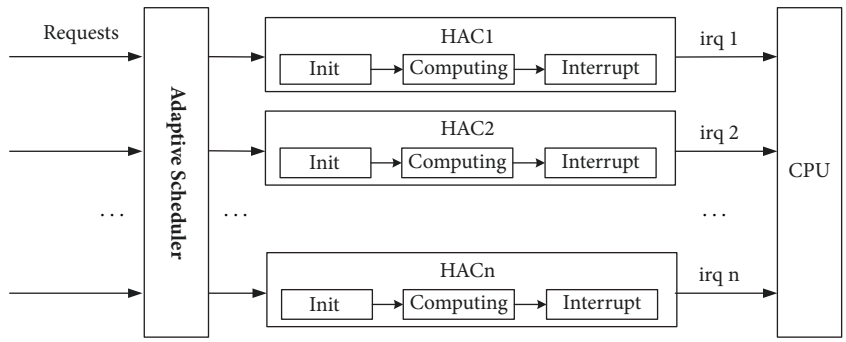


Figure 4: Interrupt processing flow without interrupt integration.

provides API for the kernel to receive delivered data from CryptoDev and constructs them as the data block that HACs could resolve. Then driver delivered prepared encryption data to hardware engines through PQs. Once interrupt signal from HACs is detected, the driver will get computing results and return to the upper layer for further transmission.

Hardware accelerators are computing units for encryption/decryption. These components get tasks from processing queues and could compute in parallel. HACs ask for interrupt if encryption computing completed informing upper layer for results return.

3.3. Dynamic Interrupt Aggregation. The interrupt is useful for results return in acceleration systems. However, we find that the interrupt processing also induces lots additional overhead in Web Server applications if the concurrent requirements are high. As illustrated in Figure 4, computing tasks are assigned to HACs through the Adaptive Scheduler (please refer to Section 4 for the detail of Adaptive Scheduler). Accelerator works independently and one interrupt occurs for each encryption task to inform that the results are ready if computing is complete. We analyzed the overhead induced by each interrupt, which includes the following:

(1) CPU will invoke interrupt processing and need one context switch, if interrupt is generated. This overhead can be denoted as C_{cs}.

(2) Interrupt handling routine reads encrypted results and responds to the peripherals. The cost induced by these operations is expressed as C_{irq}.

(3) Once interrupt handling routine is complete, the upper application will return and continue the following processing, which generates one context switch again. This cost is recorded as C_{cs}.

Based on the analysis above, the expense induced by one interrupt could be calculated as $C_{interrupt} = 2\ C_{cs} + C_{irq}$. If the requirements for hardware encryption are concentrative, the interrupt frequency will be very high. High frequency of interrupt would incur considerable system cost for the whole ACSA system. Therefore, to reduce the overall system cost, we proposed adaptive interrupt aggregation in this work.

As shown in Figure 5, instead of one interrupt with one encryption, hardware accelerators invoke interrupt if multiple encryption tasks are executed. Similarly, all the waiting processing in the upper layer is wakened up through the aggregated interrupt to return the ciphertext. For the scenario with fewer requirements, we configure a timeout threshold to confirm the interrupt processing correctly.

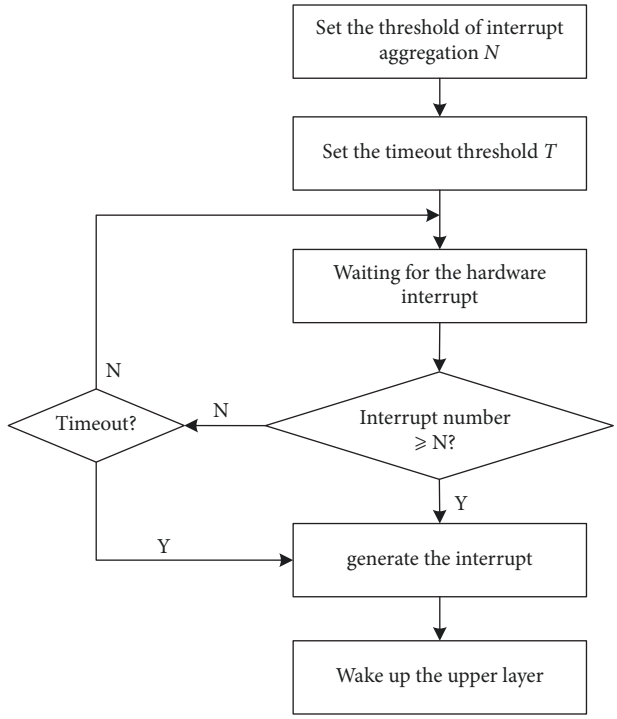


Figure 5: Working flow of adaptive interrupt aggregation.

An aggregation query is extended in the improved interrupt handling to check the completion of hardware encryption. For all the detected tasks, only one interrupt will be invoked to inform the HAC drive that N encryptions are finished with accelerators. Here, N is the number of finished encryption tasks during the query period. For a better trade-off between overhead and response delay, the threshold N is adjustable according to the system load. If the working load is high, aggregation threshold N will be increased automatically to decrease the interrupt frequency. Otherwise, N is decreased to improve the interrupt processing frequency, thus decreasing the response delay.

Before interrupt aggregation, system overhead for n interrupts will be

$$C_1 = n \times \left(2C_{cs} + C_{irq}\right). \quad (1)$$

With proposed interrupt aggregation, the overhead is reduced to

$$C_2 = 2C_{cs} + n \times C_{irq}. \quad (2)$$

Through the calculation formula, we can see an obvious overhead reduction for context switch with the interrupt aggregation. This reduction trend is strengthening with the increasing of interrupt number n for a certain application scenario.

4. Adaptive Scheduler Based on HW-SW Codesign

As a heterogamous computing platform, ACSA includes multiple processing cores and hardware accelerators. To take full advantages of both accelerators and CPUs, the Adaptive Scheduler is designed to realize optimal resource allocation. The scheduler calls OpenSSL lib or invokes accelerators for crypto computing according to the different features of application requests. However, as we know, the invocation of hardware accelerators also induces external management overhead for a specific system. Furthermore, in both ARM and X86 based architectures, most systems support cryptographic instructions, such as AES-NI. Therefore, we firstly analyze the working flow in detail to explore the difference between cost and performance for the encryption mode of AES-NI and accelerators. Through this way, we are able to gather statistic of the time waste for each segment in the working flow and figure out the hardware offloading conditions for SSL/TLS based connections.

4.1. Overhead Analysis for Workload Offloading. To guarantee the universality, we adopt the standard Crypto API framework in kernel for the invocation of hardware accelerators, which includes the overhead for Mode Switch and Context Switch. The detailed processing steps and overhead are concluded as follows:

(1) Passing the key through *ioctl* and creating crypto session: *ioctl (ctx->cfd, CIOCGSESSION, &ctx->sess)*, this process will induce twice Mode Switches (entering into the kernel, and back from the kernel to the user space).

(2) Passing the request data (original data) through *ioctl* to the driver in kernel, *ioctl (ctx->cfd, CIOCCRYPT, &cryp)*, this step will generate another Mode Switch.

(3) After the kernel submits the request to the driver, it calls function *waitfor()* and waits for the complementation of hardware computing. During this period, current user program enters a sleep state, and the kernel will invoke another process. This stage will cause one context switch.

(4) The hardware accelerator performs crypto computing asynchronously and interrupts after the execution is complete. The interrupt processing results in a context switch.

(5) Once the interrupt processing is completed (generated in step (4)), function *complete()* will be invoked to notify the user program that the request has been executed. Then the kernel will schedule the submission procedure of the user space for next scheduling cycle; this will produce a context switch.

(6) The process that submits the request gets encrypted results and returns back to the user space; i.e., the second *ioctl* returns, generating a mode switch between the user space and the kernel space.

Based on the above analysis, we decomposed the overhead for HAC invocation into 3 parts: hardware initialization time: t_{init}, accelerator execution time: t_{exec}, and the time for interrupt processing and mode switch: t_{irq}. Therefore, the overhead for hardware acceleration could be denoted as $t_{hw} = t_{init} + t_{exec} + t_{irq}$.

To figure out the hardware offloading conditions for SSL/TLS based applications, we tested the execution time of three working flows for different block sizes: t_{sw}, the time needed for software computing with AES-NI (denoted as SW-NI), t_{hw}, the time needed for hardware encryption with accelerators (denoted as HW), and the time needed with NULL CIPHER, in which, the SW-NI means all the SSL/TLS connection and encryption operations are performed through CPUs with available OpenSSL library. While in HW working mode, the crypto computing is offloaded to hardware accelerators, and the access to hardware engines for software is realized through CryptoDev methodology (please see Section 3 for details). The NULL CIPHER mode means only do *scatter_walk* for incoming scatter list and without any crypto operation. The time for *scatter_walk* describes the cost of mode switch between the user space and the kernel, and the scan for scatter list. This overhead is essential to invoke hardware accelerators. NULL CIPHER mode is utilized to test the general cost of Crypto API framework and calculate the corresponding overhead of t_{init} and t_{irq}.

We utilize the standard benchmark *speed* to get overhead time in different working mode. For a certain block size, we record total execution time for 1000 times encryption and calculate the average value.

4.2. Request Filtering and Data Aggregation. As we analyzed before, the utilization of acceleration induced additional overhead. To make full advantages of hardware acceleration engines, request filtering is applied firstly in the Adaptive Scheduler. As illustrated in Figure 6, if the data size is small, the scheduler chooses software encryption with AES-NI to avoid additional cost. Otherwise, hardware accelerators are invoked through CryptoDev. To further reduce the invocation overhead, request aggregation could be followed if hardware acceleration is adopted.

The definition for block size threshold depends on the cost comparison. Only if the additional overhead to invoke hardware is less than SW-NI, the offloading is practical. That is to say, the following criteria should be satisfied:

$$\begin{gathered} t_{sw} > c_{hw} = t_{init} + t_{irq} \\ \text{i.e., } T_{diff} = t_{sw} - \left(t_{init} + t_{irq}\right) > 0 \end{gathered} \quad (3)$$

Table 1: The running time for accelerator invocation and AES-NI.

Block Size (Bytes)	Execution time with AES-NI: t_{sw} (us)	Execution time with hardware acceleration: t_{hw} (us)		
		Initialization: t_{init}	Encryption: t_{exec}	Invocation Cost: t_{irq}
64	0.098	2.188	1.100	7.552
128	0.156	1.816	1.180	7.814
256	0.282	1.546	1.360	7.854
512	0.517	1.460	1.660	7.680
1024	0.985	1.918	2.300	7.282
2048	1.936	1.614	3.600	7.546
4096	3.838	1.874	6.140	7.506
8192	7.653	1.774	11.280	7.806
16384	15.259	1.866	21.500	7.984
32768	30.479	2.034	42.000	8.636
65536	60.902	2.128	82.940	10.212

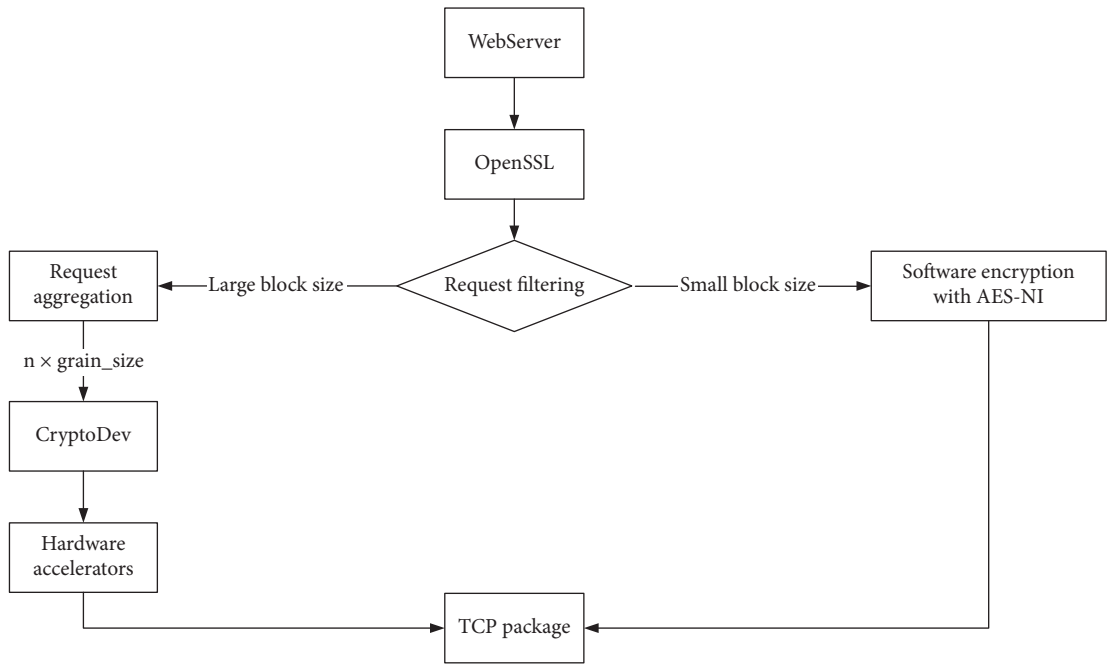


Figure 6: Working flow of Adaptive Scheduler.

Therefore, we need to test the execution time of software computing t_{sw} and accelerator invocation cost c_{hw} in the condition of the various block size and confirm the threshold according to the difference between the two parameters.

Here, we take AES-128-CBC as an example to introduce the threshold determination for request filtering.

According to the tested data as Table 1, we can draw out the difference between t_{sw} and c_{hw} in the condition of variable block size (as shown in Figure 7).

As we can see from the trend in Figure 7, when block size < 16 KB, $t_{sw} - c_{hw} < 0$, the running time with AES-NI is smaller. While block size $\geqslant 16$ KB, $t_{sw} - c_{hw} > 0$, the overhead of software computing is bigger than accelerator encryption. Therefore, we confirm the threshold in this example for workload offloading as 16 KB. If the request block size is larger than 16 KB, the scheduler will invoke hardware for accelerations; otherwise, only software working mode will be adopted.

If hardware acceleration is adopted, OpenSSL will do SSL processing for original data and then send the request to accelerators through CryptoDev and hardware driver. For each request, OpenSSL will do segmentation if the request data is larger than grain size. Here, the grain size is defined as the unit of OpenSSL processing block. For each grain block, OpenSSL will do compression, MAC, adding explicit IV, and padding firstly, and then delivery encapsulated grain

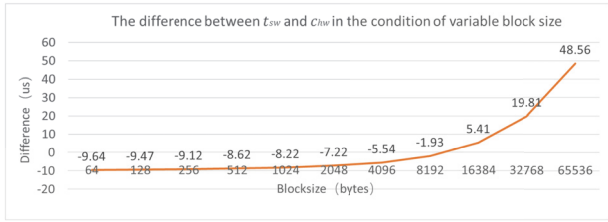


FIGURE 7: The difference between t_{sw} and c_{hw} in the condition of variable block size.

block to the hardware drive through CryptoDev one by one. Therefore, one invocation is needed for a block encryption, and the next data block will be processed only after the former one is complete. We demonstrated the example with a grain size of 16 KB in Figure 8 following the traditional SSL processing flow. Assuming the original data size is 256 KB, and the grain size is 16KB, this request should be segmented to 8 grain blocks according to the processing unit restriction in OpenSSL. So 8 invocations will be executed for hardware encryption. As we described before, each invocation refers twice mode switch and triple context switch. Totally, the overhead for the 256 KB request will be $8\times c_{hw}$. This processing flow produces lots of additional overhead, resulting in low utilization for accelerators.

To further reduce the invocation cost of hardware accelerators, request aggregation is proposed in this literature to improve resource utilization. Through aggregation, multiple grain blocks could be encrypted through one-time accelerator invocation. The design flow for data aggregation is detailed as follows:

(1) Modify the configuration file *nginx.cfg* for Nginx, so that the function *ngx_ssl_write()* could get the data block with a length of n×grain_size.

(2) Extend the function *ssl3_write_byte()* to support processing length to n×grain_size.

(3) Strengthen function *do_ssl3_write()* with data aggregation operation. In this function, the requested data of n×grain_size follows the SSL processing flow to do compression, MAC, padding, etc., and then is aggregated as a single package for lower processing.

(4) Increase the buffer size for data write, to support aggregated data storage.

(5) Revise the encryption function, *evp_cipher()*, and issue the aggregated data block to hardware for crypto computing.

(6) Decompose the encrypted data into n segments after hardware computing completed and add TCP header for each segment.

(7) Send the n TCP packages through the calling of *ssl3_write_pending()* one by one.

As illustrated in Figure 9, for a request of n blocks (each block in grain size), proposed methodology firstly does data preprocessing (MAC, padding, etc.,), then encapsulated the n segments together, issued to hardware drive subsequently, and finally perform data encryption through one accelerator invocation. Through this way, the overhead of Mode Switch and Context Switch will reduce to $1/n$ compared with the original solution and greatly improve the utilization of hardware engines.

5. Maximize Resource Utilization with Minimal Management Cost

For most Web Server applications, high concurrent requests need to be processed in time; that is why we integrate multiple CPUs and accelerators in a single server. As a heterogeneous platform with both CPUs and hardware engines, it is important to give the best play to respective computing superiorities. However, how to make full utilization of hardware engines with least system cost? How to schedule the concurrent multiprocess to get a best trade-off between performance and overhead? To solve these problems, we proposed MM strategy for resource allocation from an overall point of view. The core of the methodology is to maximize resource utilization with minimal management cost and get a best system performance through hardware and software codesign.

For a better description, we defined referred parameters firstly as shown in Table 2.

The MM allocation strategy is shown as Figure 10 and Algorithm 1. Assuming there are N CPUs available in the system, in which N_{hw} CPUs are responsible for accelerators invocation, so the number of CPUs for software encryption should be $N_{sw} = N - N_{hw}$. If there are P_total available processes in the system, in which the number of processes activated for hardware engines is P_{hw}, and the number of processes used for software computing is P_{sw}.

On a condition of the same CPUs number, if the maximum bandwidth of hardware encryption is recorded as B_{hw}, and the maximum bandwidth with AES-NI is denoted as B_{sw}, then the theoretical maximum bandwidth with hardware and software codesign should be

$$B_{max} = B_{hw} + (N - N_{hw}) \times B_{sw} \quad (4)$$

The core of MM algorithm is to find an allocation strategy with parameters N_{hw} and P_{hw}, so as to get a best system performance with available resource through hardware and software codesign. The parameter N_{hw} found by MM algorithm should be the least number of CPUs needed for accelerator management, while the parameter P_{hw} found by MM should be the least number of processes needed. Through MM strategy, we can make full utilization of hardware accelerators with least system resource occupation. Thus, the remaining CPUs are free for software computing with AES-NI, or do other operations in need [29, 30].

The major steps for MM strategies are as follows:

(*S1*) Activate i CPUs (i=1, 2, . . ., N), i processes for software encryption, and get the maximum encryption bandwidth at a condition of CPU loaded completely: SP_i.

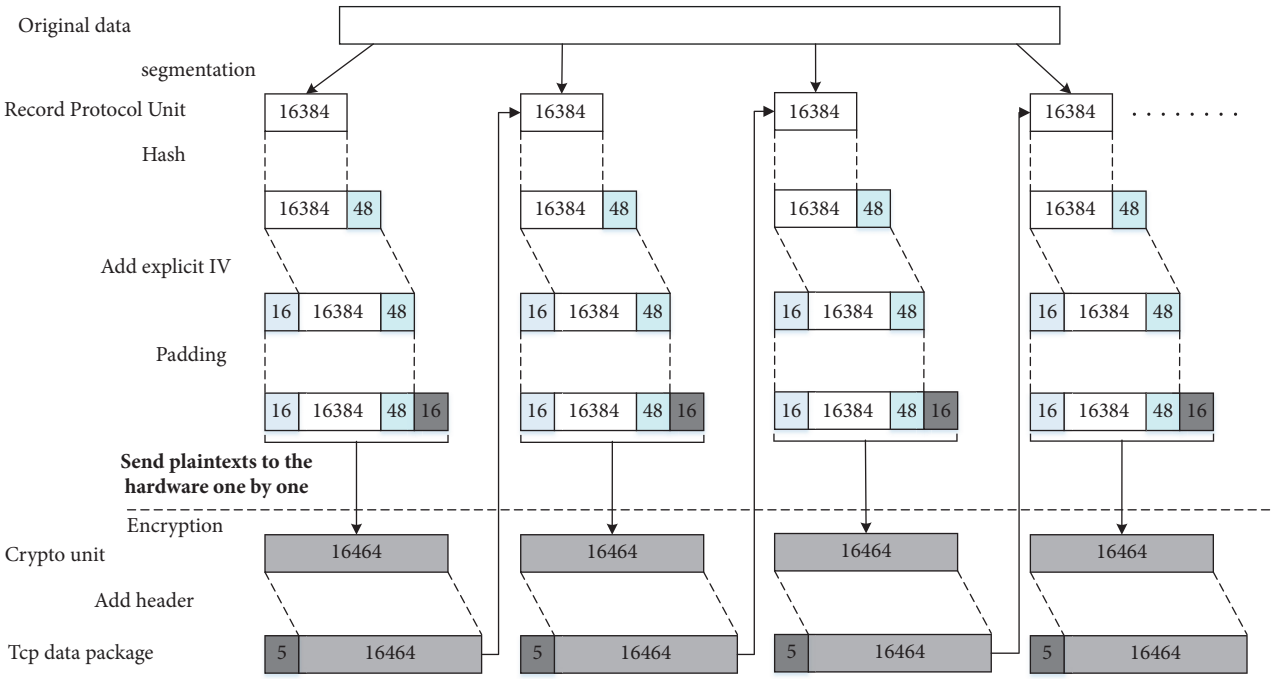


Figure 8: Existing processing flow for a 64 KB data without data aggregation.

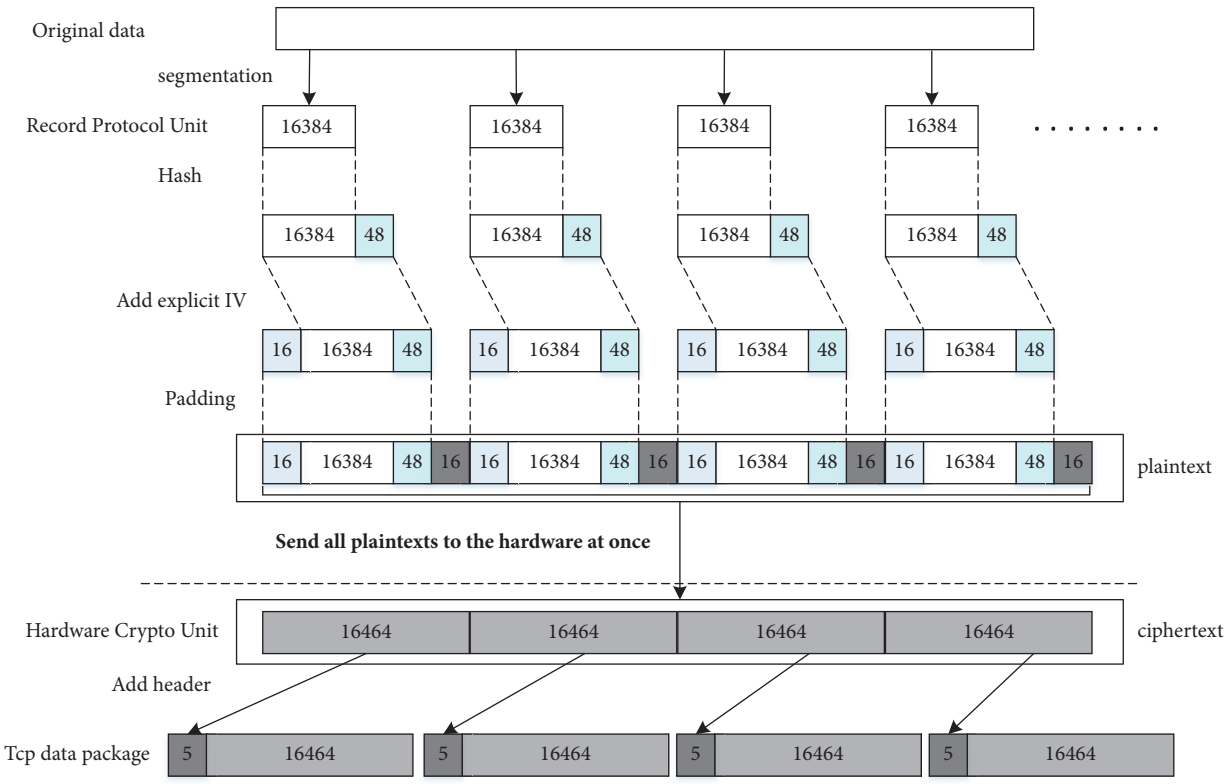


Figure 9: Proposed processing flow for a 64 KB data with data aggregation.

Table 2: Parameters for MM algorithm.

Parameter	Description
N	The number of available CPUs in the system
N_{sw}	The number of CPUs used for software encryption with AES-NI
N_{hw}	The number of CPUs responsible for the processes of hardware invocation
P_{sw}	The processes invoked for software encryption with AES-NI
P_{hw}	The processes invoked for hardware encryption
P_{total}	The total number of active processes
HP_{i_j}	The maximum bandwidth with hardware encryption, at the condition of i CPUs and j processes, and completely loaded
SP_i	The maximum bandwidth with software encryption, at the condition of i CPUs and i processes, and completely loaded
P_{i_j}	The bandwidth difference between hardware encryption and software encryption ($HP_{i\_j} - SP_i$)

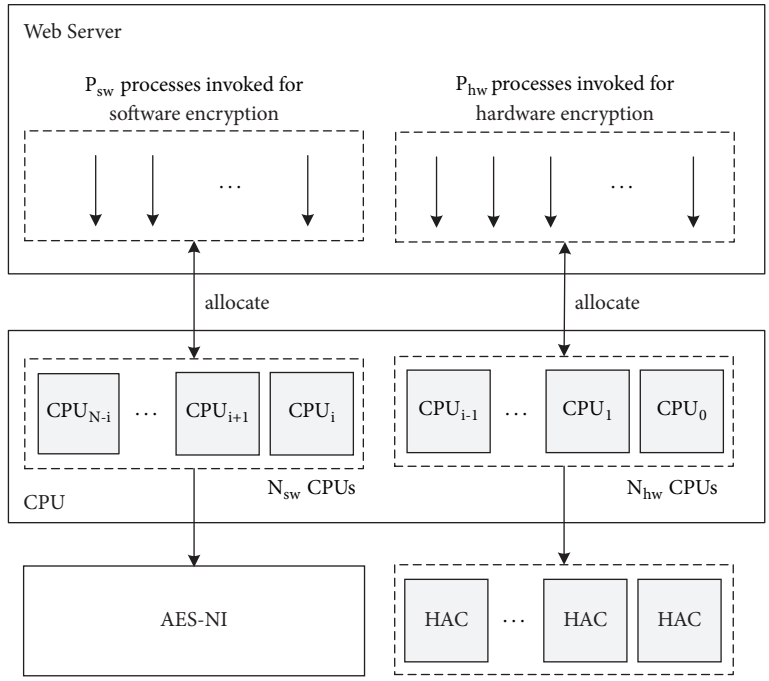


Figure 10: CPUs and processes allocation with MM strategy.

(*S2*) Activate i CPUs (i=1, 2, . . ., N), increase the number of processes j for hardware invocation, and find the maximum encryption bandwidth at a condition of CPU loaded completely: HP_{i_j}.

(*S3*) Calculate the performance difference between software computing and hardware encryption: $P_{i\_j} = HP_{i\_j} - SP_i$.

(*S4*) For the cases: i=1, 2, . . ., N, follow the steps (*S1*), (*S2*), and (*S3*) one by one, and get the value P_{i_j} at the different number of CPU.

(*S5*) Find the maximum P_{i_j} of (S4), $\max(P_{i\_j}) \in P_{i\_j}$ (i=1, 2, . . ., N). Through $\max(P_{i\_j})$, the parameters can be determined: the number of CPU used for accelerator invocation, $N_{hw} = i$; the number of processes for hardware encryption, $P_{hw} = j$; the number of CPU for software encryption: $N_{sw} = N - N_{hw}$, and the corresponding number of processes $P_{sw} = N_{sw}$. The total processes should be activated as $P_{total} = P_{sw} + P_{hw}$.

To introduce MM algorithm more clearly, we take AES-128-CBC as an example for the exploration of parameter i, j. Assuming the N as 16 in this example. The detailed process followed by MM is as follows:

(*S1*) Adopt the working mode as software encryption through Adaptive Scheduler in ACSA. All the encryption requests are processed through AES-NI. The working flow is illustrated as Figure 11(a).

(*S2*) Activate one CPU and one process of ACSA, augment the workload to make CPU completely loaded, and get the maximum encryption bandwidth as 1.01 GB/s.

(*S3*) Activate 2~16 CPUs orderly, corresponding to 2~16 processes separately. Similar to (*S2*), maximum encryption bandwidth can be explored at the different

Input: N: The number of available CPUs in the system.
Output: N_{sw}: The number of CPUs used for software encryption with AES-NI;
N_{hw}: The number of CPUs responsible for the processes of hardware invocation;
P_{sw}: The processes invoked for software encryption with AES-NI;
P_{hw}: The processes invoked for software encryption with AES-NI;
P_{total}: The total number of active processes.
/* test for the maximum bandwidth with AES-NI in different CPU number, general the process number is equal to the CPU number */
(1) for CPU number from 1 to N do
(2) calculate SP_i /* SP_i is the maximum bandwidth when the CPU number is i */
(3) end for
/* test for the maximum bandwidth with hardware in different CPU number and process number */
(4) for CPU number from 1 to N do
(5) for process number from 1 to 32 do
(6) calculate HP_{i_j} /*HP_{i_j} is the maximum bandwidth when the CPU number and process number is i and j */
(7) end for
(8) end for
/* Calculate the max performance difference between AES-NI computing and hardware encryption */
(9) let maxdiff $\longleftarrow$ P_{1_1}.
(10) for CPU number from 1 to N do
(11) for process number from 1 to 32 do
(12) $P_{i\_j} \longleftarrow (HP_{i\_j} - SP_i)$.
(13) if (maxdiff < P_{i_j}) then
(14) maxdiff$\longleftarrow$ P_{i_j}.
(15) $N_{hw} \longleftarrow i$.
(16) $P_{hw} \longleftarrow j$.
(17) $N_{sw} \longleftarrow$ (N-i).
(18) $P_{sw} \longleftarrow N_{sw}$.
(19) $P_{total} \longleftarrow (P_{sw} + P_{hw})$.
(20) end if
(21) end for
(22) end for

Algorithm 1: The strategy for MM algorithm.

Table 3: The number of processes P_{hw} at different number of available CPUs when best crypto performance achieved.

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P_{hw}	12	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16

number of CPUs and processes. We conclude the SP_i (i = 1, 2. . .16) in Figure 12.

(*S4*) Adopt the working mode as hardware encryption through Adaptive Scheduler in ACSA. All the encryption requests are processed through hardware accelerators. The working flow is illustrated as Figure 11(b).

(*S5*) Activate one CPU and one process of ACSA, augment the workload to make CPU completely loaded, and get the maximum encryption bandwidth as 7.52 GB/s in software working mode.

(*S6*) Still keep one CPU activated, invoke 2~n processes orderly. Record the maximum encryption bandwidth when CPU completely loaded at different cases. If the bandwidth with n processes invoked is less than or equal to the bandwidth of n-1 processes invoked, the parameter j can be confirmed as n-1, and the test can be stopped.

(*S7*) Activate 2~16 CPUs in turn, and follow steps (*S5*) and (*S6*) to confirm the number of processes when maximum encryption bandwidth is achieved. We recorded the number of processes at different number of CPUs as Table 3, and the corresponding bandwidth is shown in Figure 12.

(*S8*) Compute the performance difference between software and hardware encryption; the results are recorded as in Figure 13.

From Figure 13, we can find the maximum difference in the case of N =1, P_{hw}= 12, i.e., the number of CPU is 1, and the corresponding processes are 12. Based on the test results, 27 processes should be invoked for the system, in which 12 are scheduled for hardware management, and remaining 15 processes could be allocated for software encryption with AES-NI if no other works are in need.

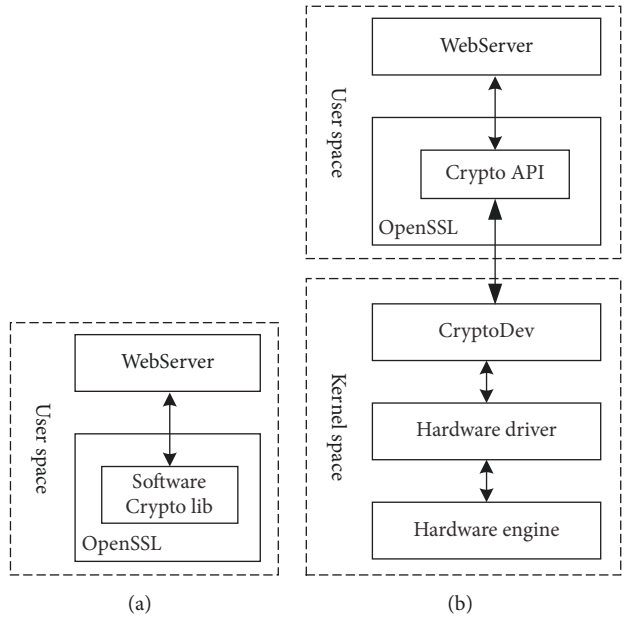


Figure 11: The working flow of (a) software encryption and (b) hardware encryption.

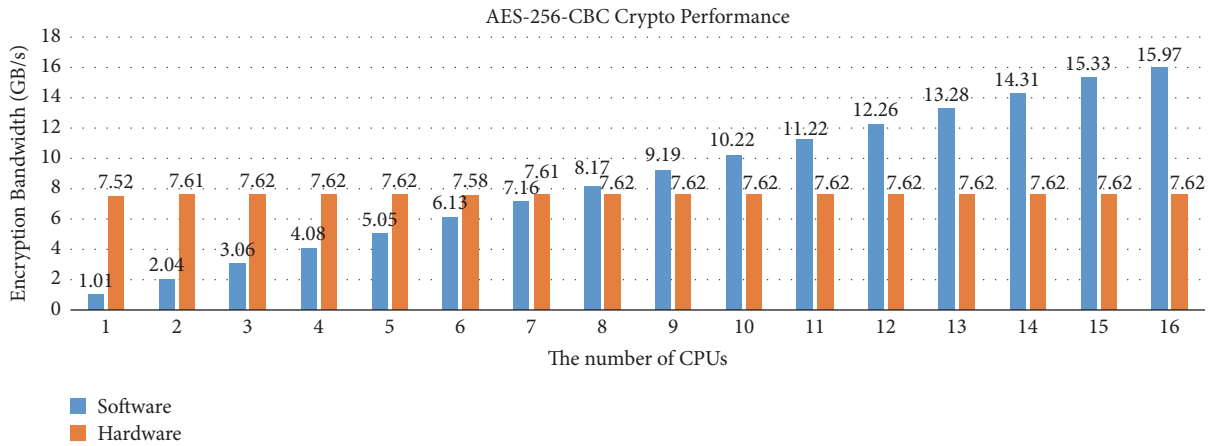


Figure 12: The encryption performance with hardware/software working mode at different CPU number.

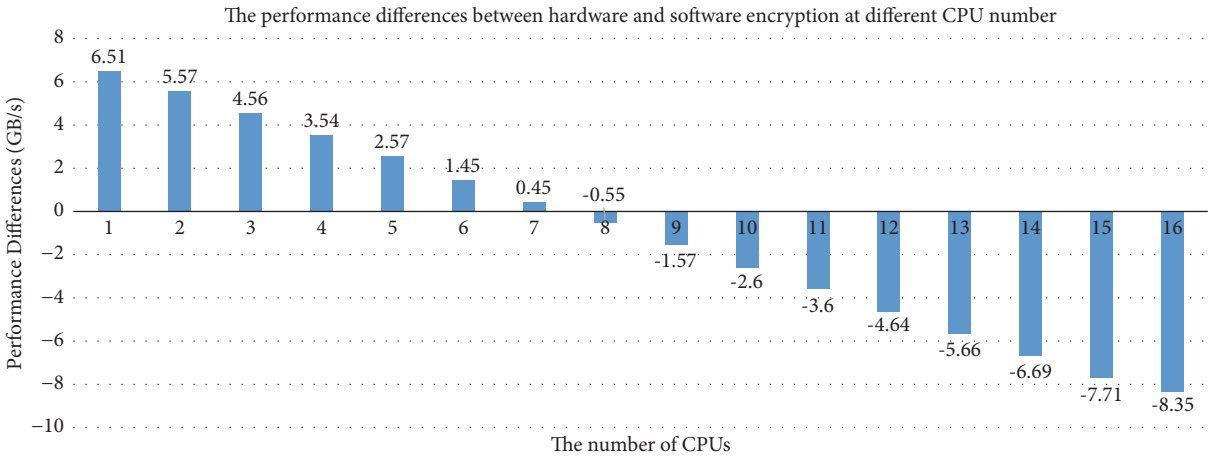


Figure 13: The performance differences between hardware and software encryption.

Table 4: Hardware environment of testing.

Hardware	Number	Configurations
ARM Server	2	CPU: ARM Cortex-A57,16 cores Main frequency: 2.1 GHz; Memory: 128 GB
Network Card	4	10 Gbps each
Net cable	4	2 Gigabit cables (Category 7A); 2 10 Gbps fiber cables

Table 5: Software environment of testing.

Software	Configuration / Software Version
Operating System	Linux-4.1.27
OpenSSL	OpenSSL-1.0.2j
Nginx	Nginx-1.11.6
Benchmark	ab (apache benchmark)

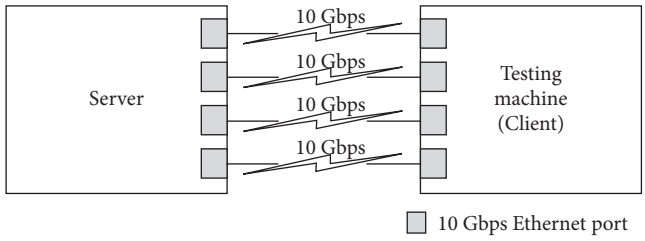


Figure 14: Testing network (CASC acted as Server, the Testing machine as Clients.)

6. System Test and Analysis

To reflect the real working environments, as shown in Figure 14, we establish a test platform for ACSA with 40 Gbps network bandwidth. As shown in Table 4, we deployed 2 Web Servers for testing; one is used as ACSA crypto system to response HTTPS accesses, while another one is utilized as a client to generate testing workloads. Both machines are equipped with 16 ARM processing units, whose main frequency is 2.1 GHz, and memory size is 128 GB. To satisfy the high concurrent requirements from mass clients, we established the networking environments with 4 Network Cards, and each contributes to 10 Gbps bandwidth.

As illustrated in Table 5, we adopt Nginx [31] as Web Server, which uses an asynchronous event-driven approach to handle requests. The operating system is adopted as Linux-4.1.27 and is extended to support HAC access. OpenSSL-1.0.2j is utilized to perform software encryption through cryptographic library libcrypto. Adaptive Scheduler and MM algorithm are also integrated into OpenSSL for efficiently utilization of accelerators. Ab (apache benchmark) [32] is a widely used testing toolset for website performance. We choose this benchmark for end-to-end testing in a network.

6.1. Testing Methodology. To guarantee the correctness and completeness, proposed ACSA is evaluated from two working levels:

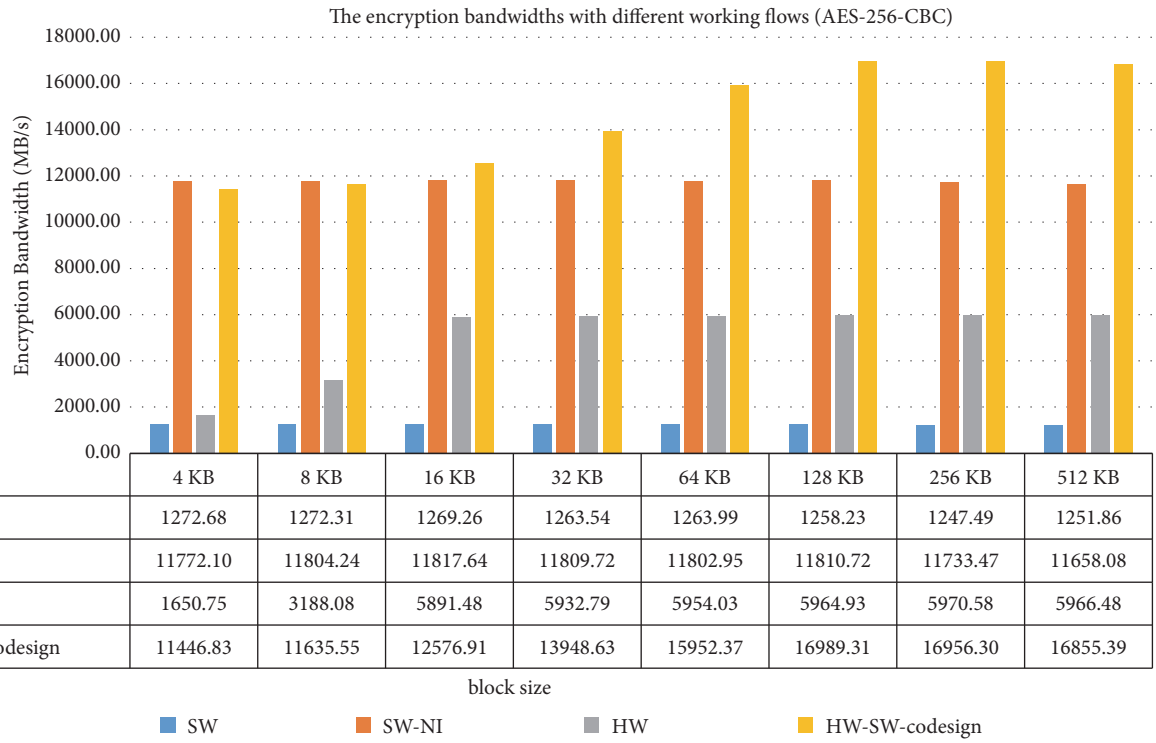
(1) OpenSSL Testing on the Server Side. This testing is performed through the standard benchmark, *speed* in OpenSSL. We could detect the maximum encryption bandwidth for a single algorithm with *speed*. The tested encryption algorithm includes AES-256-CBC and 3DES-CBC, in which AES-CBC could be accelerated through AES-NI, while 3DES-CBC is not supported by AES-NI. To explore the performance for different configurations and data characters, we vary the number of processes, the block size for encryption, and the number of available ARMs. The performance is evaluated through MB/s in this testing, which indicates the amount of data that can be encrypted per second.

(2) End-to-End Testing for Full System. End-to-end testing is utilized to evaluate the performance of HTTPS service that can be provided by ACSA. We use the testing machine to generate HTTPS requirements from different clients. Through established 40 Gbps networking, we could increase the workload to simulate high concurrent accesses in real application scenarios. Standard toolset *ab* is adopted for workload testing, and the performance indicator is RPS (Request per Second). For further analysis, we choose the cipher suite based on the same testing algorithms in OpenSSL. Two cipher suites are tested in our experiments: ECDHE-RSA-AES256-SHA384 and ECDHE-RSA-DES-CBC3-SHA. Different with OpenSSL testing, the tested data are web pages instead of data blocks. We tested 7 different web pages to see the performance for different data size. We take full advantages of 4 network cards to provide 32 *ab* processes, so as to confirm the testing pressure with a high workload.

6.2. OpenSSL Testing. We tested the encryption bandwidth for both AES-256-CBC and 3DES-CBC in this section. For each experiment, we evaluated maximum encryption bandwidth with different block sizes, including 4K, 8K, 32K, 64K, 128K, 256K, and 512K. All the results are presented as MB/s (Megabytes per second) in Figures 15 and 16.

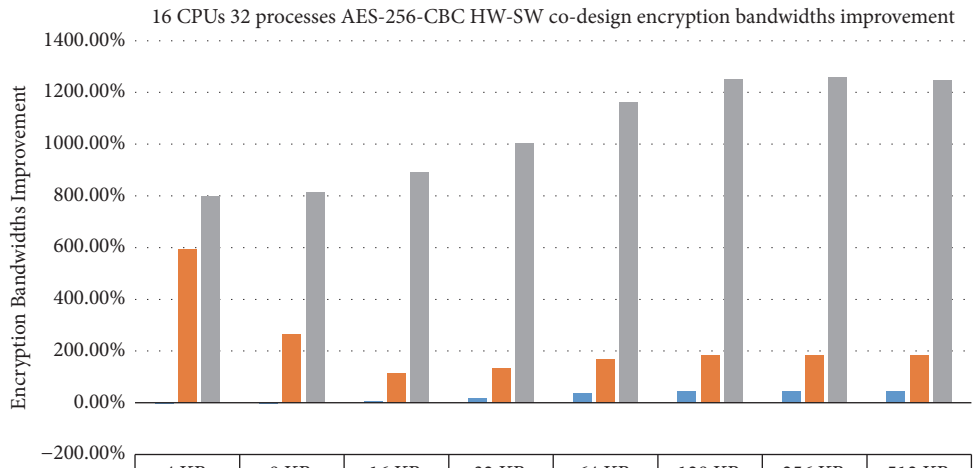
6.2.1. AES-256-CBC. We tested and compared 4 working flows for AES-256-CBC: (1) software encryption with CPU, the crypto computing is executed through OpenSSL libcrypto. We denoted this working flow as SW; (2) software encryption with AES-NI, the software computing is accelerated through special instruction set, AES-NI. This working flow is denoted as SW-NI for clarity; (3) hardware encryption with accelerators, but without adaptive scheduling and MM strategy, and this working mode is denoted as HW; and (4) hardware encryption with accelerators, with our proposed optimization method: adaptive scheduling and MM strategy. This testing is denoted as HW-SW codesign; for fairness, the adaptive interrupt aggregation is applied to all the testing flows.

Figure 15 showed the encryption bandwidth with four different working flows. To better present the performance improvement with our proposed methodology, we concluded the bandwidth comparison with HW-SW codesign and AES-NI in Figure 16, in which, the encryption flow with AES-NI is the best situation with software encryption.



	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB
SW	1272.68	1272.31	1269.26	1263.54	1263.99	1258.23	1247.49	1251.86
SW-NI	11772.10	11804.24	11817.64	11809.72	11802.95	11810.72	11733.47	11658.08
HW	1650.75	3188.08	5891.48	5932.79	5954.03	5964.93	5970.58	5966.48
HW-SW-codesign	11446.83	11635.55	12576.91	13948.63	15952.37	16989.31	16956.30	16855.39

Figure 15: Encryption bandwidths with SW, SW-NI, HW, and HW-SW codesign for algorithm AES-256-CBC.



	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB
HW-SW co-design vs. SW-NI improvement	-2.76%	-1.43%	6.42%	18.11%	35.16%	43.85%	44.51%	44.58%
HW-SW co-design vs. HW improvement	593.43%	264.97%	113.48%	135.11%	167.93%	184.82%	184.00%	182.50%
HW-SW co-design vs. SW improvement	799.43%	814.52%	890.89%	1003.93%	1162.06%	1250.25%	1259.23%	1246.43%

Figure 16: Performance comparisons with HW-SW codesign and SW-NI/HW.

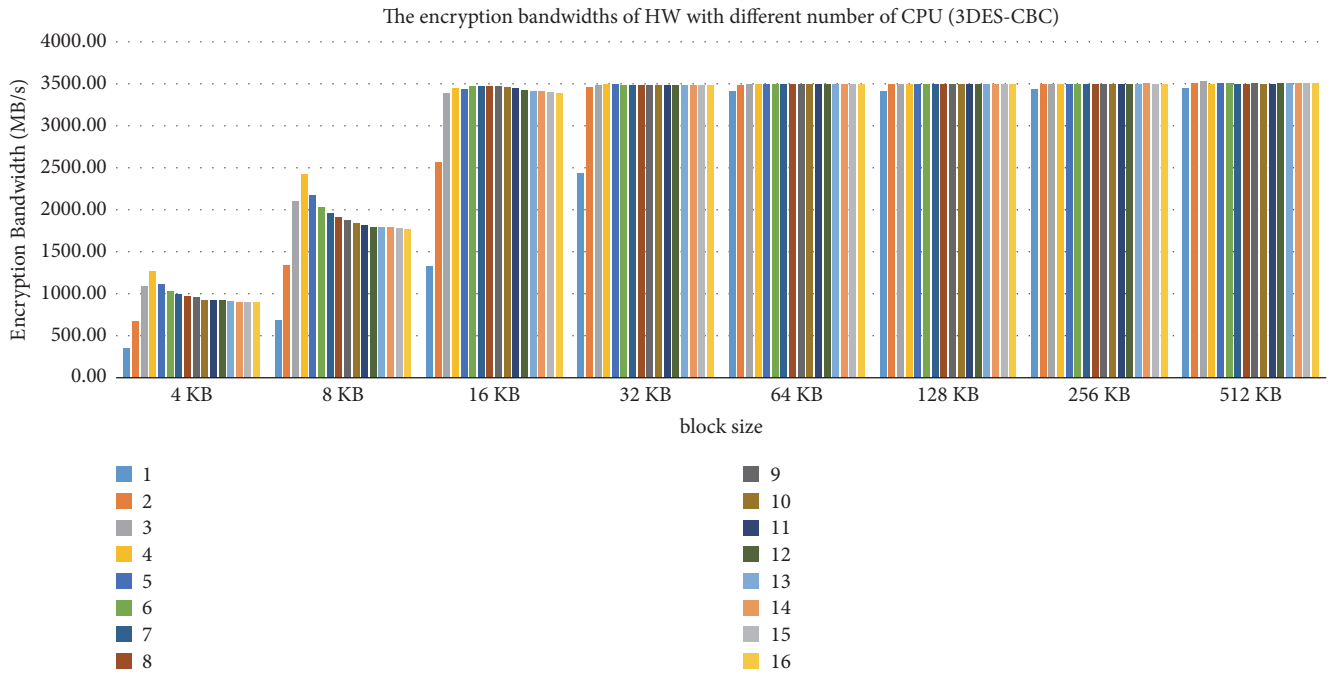


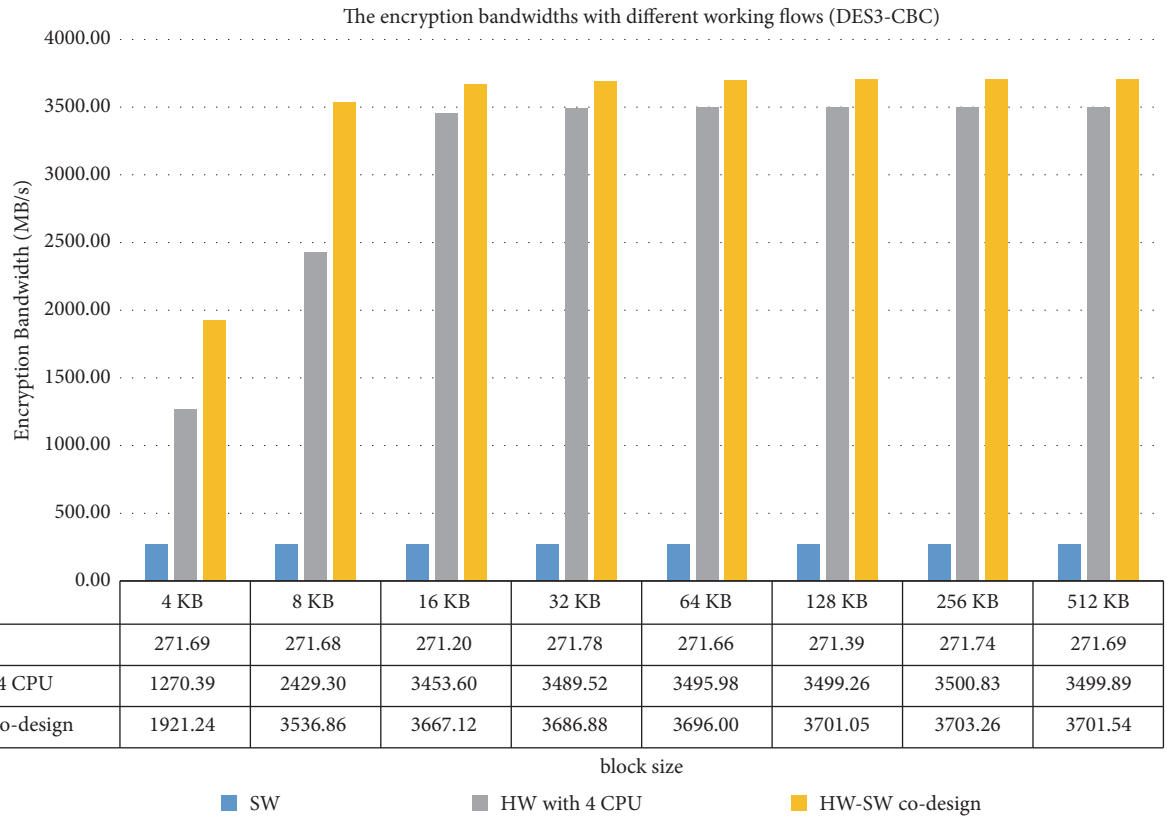
Figure 17: Encryption bandwidths with hardware engines at different number of CPUs.

As we can see from Figures 15 and 16, hardware encryption with hardware engines can get better performance compared with the crypto lib but worse than AES-NI. The reason is the invocation cost for hardware engines with context switch and mode switch as we analyzed in Section 4.1. Besides, for AES-NI, the encryption function is accelerated directly through instructions, and no additional operations are needed. Proposed HW-SW codesign is optimal almost for all the situations. We can get 799%~1246% performance improvement compared with software encryption and 593%~182% improvement compared with only hardware encryption. Even if compared with AES-NI, proposed working flow can get 18%~44% performance increase for large data blocks. If the block size is less than 16 KB, such as 4 KB and 8 KB, the maximum encryption bandwidth is a little bit lower than AES-NI. The reason is that the threshold for the data filter is configured as 16 KB in this case. ACSA will invoke AES-NI for data encryption if the data size is smaller than the threshold. Since CPU resource is needed for data size checkout and scheduling decision, the maximum encryption bandwidth is somewhat lower than AES-NI. However, this influence is decreased with the increasing of data size, on account of the reduced frequency for scheduling checkout. If the data size is bigger than 16 KB, both AES-NI and hardware engine cooperated to contribute a better overall system performance. According to the MM strategy, in this case, we allocated 13 processes for hardware encryption and 19 for software encryption. As we can see from the recorded results, through hardware and software codesign, we can get the best encryption bandwidth compared with only software or hardware encryption.

6.2.2. 3DES-CBC. Since AES-NI does not support 3DES-CBC yet, the performance with hardware engines is much better than software. It is not reasonable for doing the data filter and dynamic scheduling. Therefore, we only applied the MM strategy to make full utilization of hardware engines with minimal management cost. We firstly evaluated the encryption bandwidths with hardware engines at different number of CPUs. As shown in Figure 17, the hardware accelerators performed not so excellent when data size is small; the reason is the induced cost for many context/mode switches. For encryption with hardware engines, the larger the data blocks, the better the performance. As we can see from the results, we could get almost the best encryption bandwidth only with four CPUs.

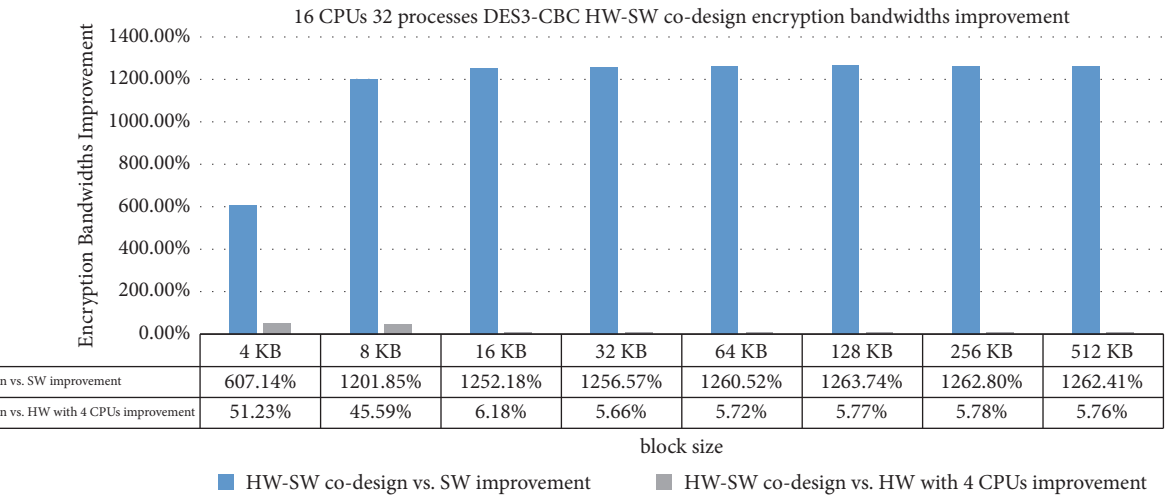
If we want to get maximum performance with both hardware and software encryption engines, we could allocate the remaining CPUs for software encryption. Here, for most block sizes, it is reasonable to utilize only four CPUs for hardware encryption. Therefore, we can take full advantages of the other 12 available CPUs for more encryption tasks. For a better presentation, we tested three different working flows: (1) software encrypting with OpenSSL crypto lib (SW); (2) hardware encryption with hardware engines, and only four CPUs are utilized (HW with four CPUs); and (3) hardware and software codesign with 16 CPUs (HW-SW codesign).

As we can see from Figure 18, even though there are 16 CPUs responsible for the 3DES encryption, the encryption bandwidth is still very small (only 271 MB/s). Since it is computation intensive, this algorithm occupies lots of system resource if there is no instruction acceleration. The encryption computation is accelerated greatly by hardware crypto engines, and the improvement is more obvious for



	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB
SW	271.69	271.68	271.20	271.78	271.66	271.39	271.74	271.69
HW with 4 CPU	1270.39	2429.30	3453.60	3489.52	3495.98	3499.26	3500.83	3499.89
HW-SW co-design	1921.24	3536.86	3667.12	3686.88	3696.00	3701.05	3703.26	3701.54

Figure 18: Encryption bandwidth for different working flows.

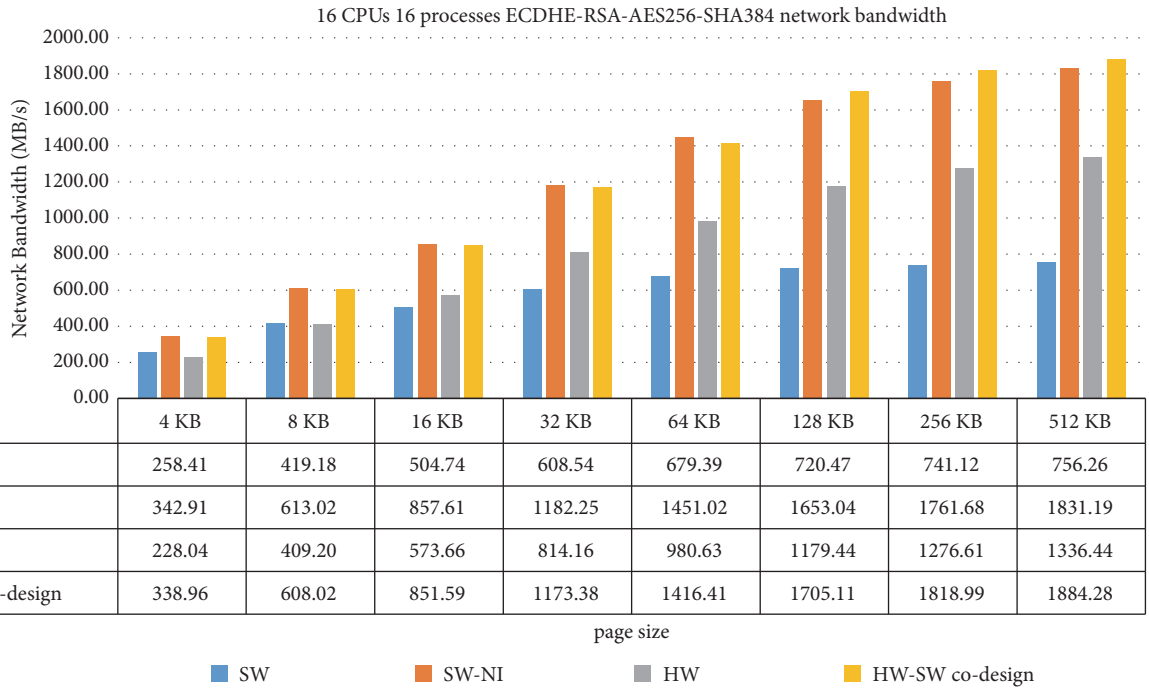


	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB
HW-SW co-design vs. SW improvement	607.14%	1201.85%	1252.18%	1256.57%	1260.52%	1263.74%	1262.80%	1262.41%
HW-SW co-design vs. HW with 4 CPUs improvement	51.23%	45.59%	6.18%	5.66%	5.72%	5.77%	5.78%	5.76%

Figure 19: Performance comparisons with HW-SW codesign, HW with 4 CPUs and SW.

larger block sizes. The reason is more invocation cost induced for smaller data blocks. The remaining CPU idle can be transformed for software encryption, and the maximum aggregate bandwidth could achieve 3703 MB/s with hardware and software codesign.

To have a clear comparison, we concluded the performance improvement in Figure 19. As we can see, there is nearly 12 times bandwidth improvement through HW-SW codesign compared with software encryption. There is only 6 times improvement for small data blocks (4 K) due to the worse utilization of hardware engines. Compared with only hardware encryption, the improvement with HW-SW codesign is not so obvious for large data blocks. The reason is the poor contribution through software encryption without AES-NI. Still, we can get 45%~51% improvement for small data blocks with HW-SW codesign, and get additional 200~



	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB
SW	258.41	419.18	504.74	608.54	679.39	720.47	741.12	756.26
SW-NI	342.91	613.02	857.61	1182.25	1451.02	1653.04	1761.68	1831.19
HW	228.04	409.20	573.66	814.16	980.63	1179.44	1276.61	1336.44
HW-SW co-design	338.96	608.02	851.59	1173.38	1416.41	1705.11	1818.99	1884.28

Figure 20: Network bandwidth with 16 processes for ECDHE-RSA-AES256-SHA384.

300 MB/s encryption bandwidth for large blocks. Since the software encryption contributes little for total crypto performance compared with hardware engines, we suggest that remaining CPU idle utilized for other important tasks such as database managements. Of course, it depends on user to decide how to make a decision about the trade-off between the CPU idle and encryption performance. For example, if it is rush hour, the user could take full advantages of both hardware and software for maximum overall performance; for normal working time with fewer encryption requirements, the user could only allocate needed management resource for HACs and try to have a best offloading.

6.3. End-to-End Testing. We adopt a pressure test to evaluate the end-to-end performance. Through pushing different workload for Web Server, we can get the encryption bandwidth and CPU idle in different working situations. We tested diverse page sizes to see the performance diversity for different request characters.

6.3.1. Cipher Suite: ECDHE-RSA-AES256-SHA384. Similar as the OpenSSL testing, we evaluated four different working flows: (1) software encrypting with OpenSSL crypto lib (SW); (2) software encryption with AES-NI (SW-NI, instruction acceleration); (3) hardware encryption using hardware engines with proposed design flow (HW); and (4) hardware encryption with hardware-software codesign (HW-SW codesign), which is further optimized with MM and data aggregation. For different request pages, we further evaluated the trade-off between bandwidth and CPU idle with 16 CPU processes and 32 processes, respectively. To better present the advantages of proposed ACSA, we showed three performance indexes in this section, RPS: request per second; MB/s: encrypted data (in Megabytes) per second, and CPU idle.

(1) 16 CPUs, 16 Processes. We showed the RPS with different working flows for cipher suite ECDHE-RSA-AES256-SHA384 in Table 7. For a clear comparison and analysis, we transit RPS to network bandwidth MB/s and presented the results in Figure 20.

As we can see from Figure 20 that HW-SW codesign can get greater network bandwidth compared with only hardware encryption. The reason is great reduction of invocation cost through proposed data aggregation and adaptive scheduling. The influence is more obvious for small block sizes. For example, the maximum improvement achieves 48.64% for case page 4 KB. If the page size is less than 128 KB, the network bandwidth with HW-SW codesign is a little bit lower than AES-NI. The reason is the cost for size detection and scheduling decision. If the page size is equal or bigger than 128 KB, the overall performance is advanced compared with only software encryption or only hardware encryption.

Although the bandwidth improvement for the proposed methodology is not obvious, even kind of lower for small requests, we still have the exploration space for further optimization with HW-SW codesign. As shown in Figure 21, even though the network bandwidth is a little bit better with AES-NI, there is no CPU idle at all. However, for HW-SW codesign, there are 12.51%~18.51% CPU idle compared with AES-NI for smaller page sizes. If the page size is larger than 64 KB, the CPU idle increases to 21.67%~23.83%, and there is also 2.85%~3.38% performance improvement.

Table 6: RPS with different working flows for ECDHE-RSA-AES256-SHA384 (16 CPUs, 16 processes).

Page Size (KB)	4	8	16	32	64	128	256	512
SW	66153.10	53654.50	32303.20	19473.40	10870.30	5763.76	2964.48	1512.51
SW-NI	87784.10	78466.20	54887.00	37831.90	23216.30	13224.30	7046.71	3662.37
HW	58378.50	52377.60	36714.30	26053.10	15690.00	9435.54	5106.42	2672.87
HW-SW co-design	86679.40	77827.00	45499.50	34357.70	22623.60	13670.90	7275.66	3770.97

Table 7: RPS with different working flows for ECDHE-RSA-AES256-SHA384 (32 processes).

Page Size (KB)	4	8	16	32	64	128	256	512
SW	65885.90	52987.80	32030.00	19391.20	10803.80	5744.14	2964.18	1503.48
SW-NI	86302.50	77827.15	53017.20	37116.10	22920.40	13092.50	7020.85	3657.35
HW	58499.10	54891.60	37946.40	26972.10	16401.90	10041.90	5486.14	2881.66
HW-SW co-design	85177.20	77211.90	46404.00	36166.60	24875.10	15347.70	8325.19	4391.45

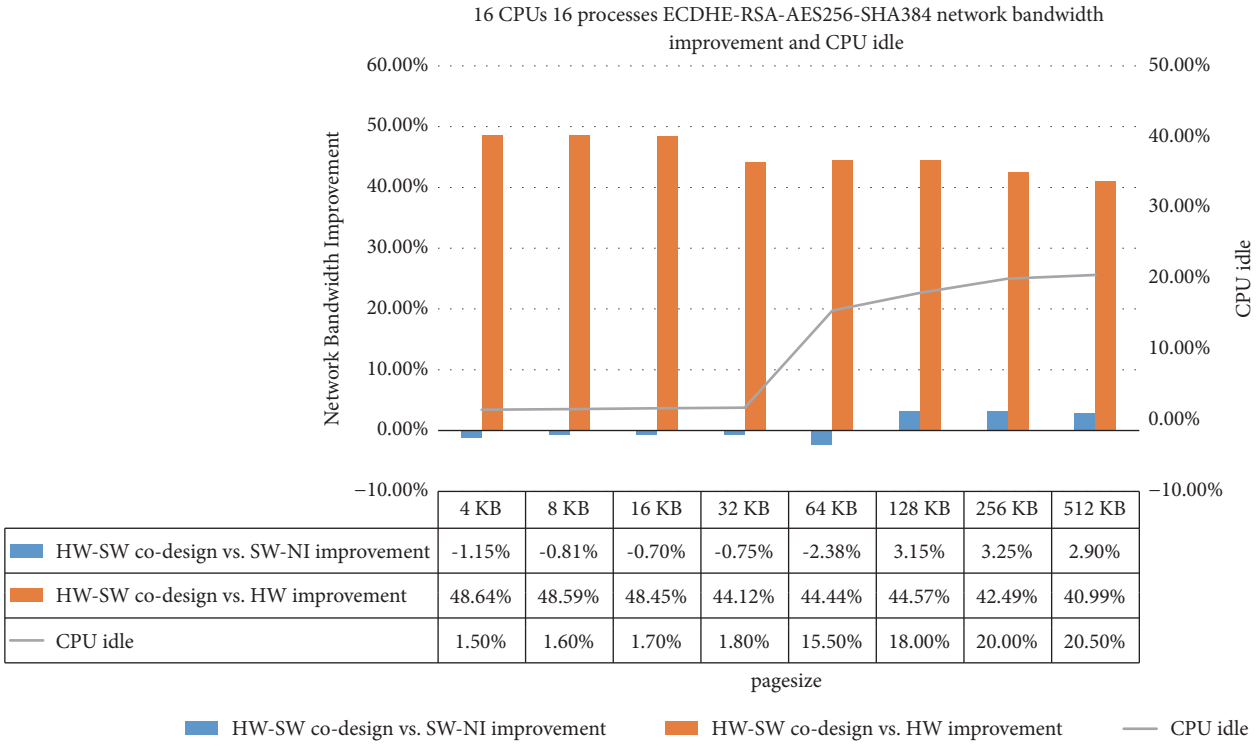
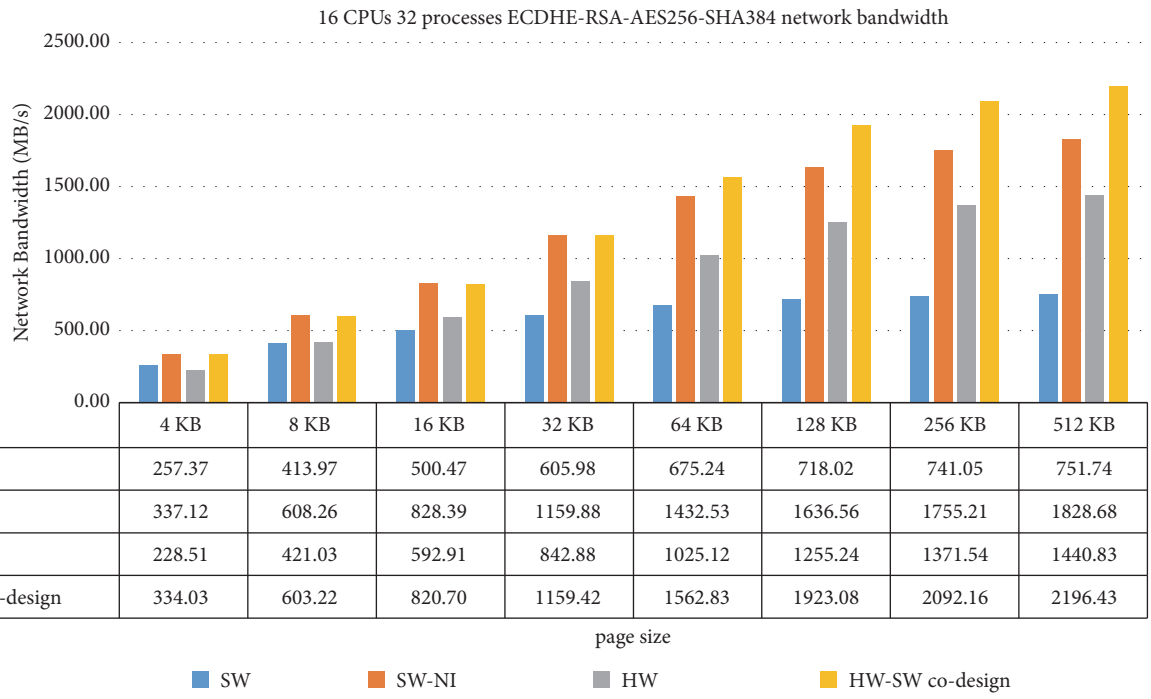


Figure 21: RPS improvement and CPU remaining with 16 processes for ECDHE-RSA-AES256-SHA384.

If we expect to get a higher network bandwidth for a better encryption performance, we could utilize the remaining CPU resource. As an example of the trade-off between CPU idle and performance, we also presented the test results with more processes as below.

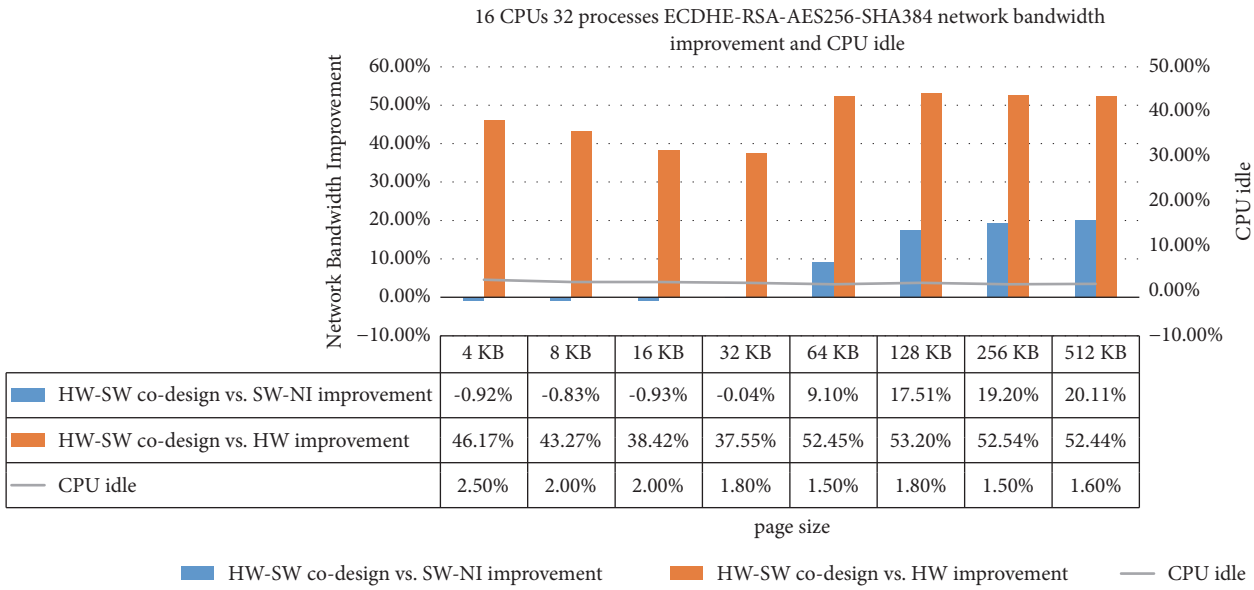
(2) 16 CPUs, 32 Processes. As we can see from Figure 21, the working flow with HW-SW codesign can get extra CPU idle compared with AES-NI. Therefore, to further explore a better performance with HW-SW codesign, we increase the number of processes to utilize the CPU idle. We tested the results with 32 processes and showed the RPS with different working flows for cipher suite ECDHE-RSA-AES256-SHA384 in Table 6. For a clear comparison and analysis, we transit the RPS to network bandwidth MB/s and presented the results in Figure 22.

As we can see from Figure 22, HW-SW codesign can get 49%~52% bandwidth improvement compared with only hardware encryption. The reason is great reduction of invocation cost through proposed data aggregation and adaptive scheduling. Nevertheless, different with 16 processes, the influence is more obvious for large block sizes since there are more CPU idles available for performance improvement. If the page size is less than 64 KB, the network bandwidth with HW-SW codesign is a little bit lower than AES-NI. The reason is the cost for data size detection and scheduling



	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB
SW	257.37	413.97	500.47	605.98	675.24	718.02	741.05	751.74
SW-NI	337.12	608.26	828.39	1159.88	1432.53	1636.56	1755.21	1828.68
HW	228.51	421.03	592.91	842.88	1025.12	1255.24	1371.54	1440.83
HW-SW co-design	334.03	603.22	820.70	1159.42	1562.83	1923.08	2092.16	2196.43

Figure 22: Network bandwidth with 32 processes for ECDHE-RSA-AES256-SHA384.

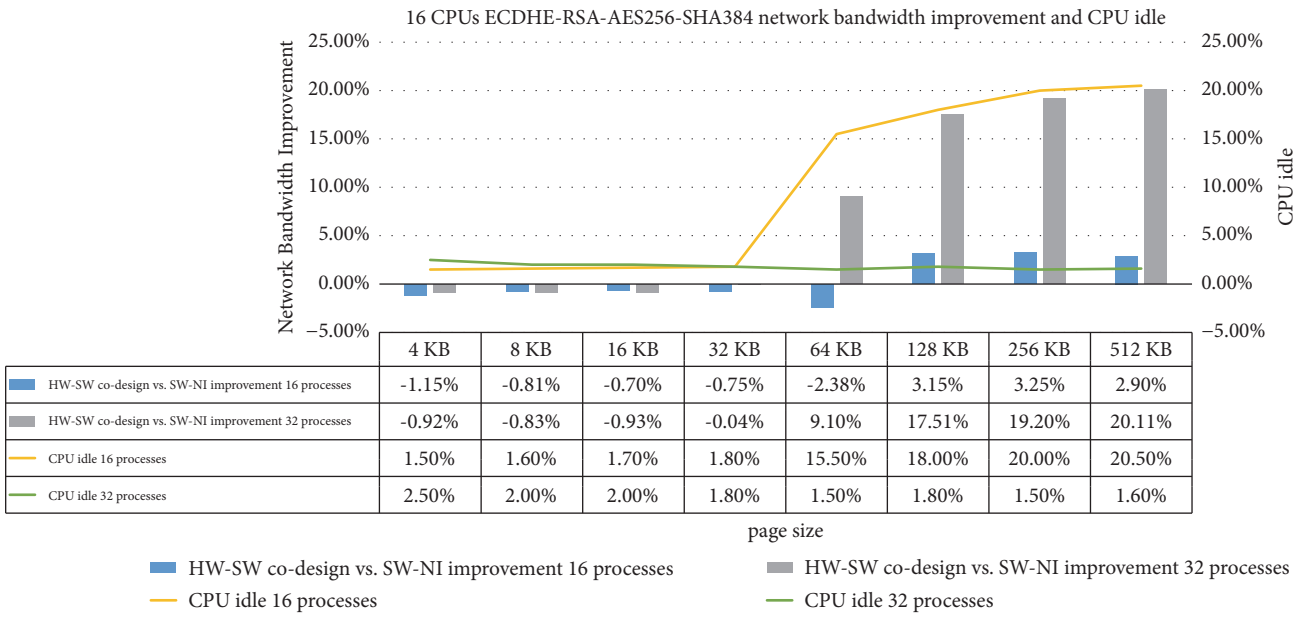


	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB
HW-SW co-design vs. SW-NI improvement	-0.92%	-0.83%	-0.93%	-0.04%	9.10%	17.51%	19.20%	20.11%
HW-SW co-design vs. HW improvement	46.17%	43.27%	38.42%	37.55%	52.45%	53.20%	52.54%	52.44%
CPU idle	2.50%	2.00%	2.00%	1.80%	1.50%	1.80%	1.50%	1.60%

Figure 23: RPS improvement and CPU remaining with 32 processes for ECDHE-RSA-AES256-SHA384.

decision. If the page size is equal or bigger than 64 KB, the overall performance is advanced compared with only software encryption or only hardware encryption.

To illustrate the advantage of HW-SW codesign more clearly, we conclude the network improvement and CPU idle in Figure 23. As we can see from the figure, for small pages, HW-SW codesign almost maintains the same performance with AES-NI. It is reasonable since we applied the request filter to choose the most suitable encryption way. Hardware engines are not so excellent compared with AES-NI due to context/switch cost. Therefore, ACSA automatically adopted software encryption with AES-NI for small pages. For large pages, the saved CPU resource through hardware and software codesign could be utilized to further improve encryption bandwidth. Even compared with AES-NI, we still get an additional 20% network bandwidth for page size 512 KB.



	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB
HW-SW co-design vs. SW-NI improvement 16 processes	-1.15%	-0.81%	-0.70%	-0.75%	-2.38%	3.15%	3.25%	2.90%
HW-SW co-design vs. SW-NI improvement 32 processes	-0.92%	-0.83%	-0.93%	-0.04%	9.10%	17.51%	19.20%	20.11%
CPU idle 16 processes	1.50%	1.60%	1.70%	1.80%	15.50%	18.00%	20.00%	20.50%
CPU idle 32 processes	2.50%	2.00%	2.00%	1.80%	1.50%	1.80%	1.50%	1.60%

Figure 24: Bandwidth improvement and CPU idle comparison between different working flows for ECDHE-RSA-AES256-SHA384.

Table 8: ECDHE-RSA-DES-CBC3-SHA RPS with different working flows (16 CPUs, 32 processes).

Page Size (KB)	4	8	16	32	64	128	256	512
SW	36334.70	25207.40	12791.60	6900.93	3589.73	1833.76	924.40	466.12
HW	58275.70	50717.80	25772.70	14525.70	7645.65	3902.54	1968.62	990.88

(3) Test Conclusion. In this section, we tested and analyzed the end-to-end performance for cipher suite ECDHE-RSA-AES256-SHA384 in diverse page sizes at different available CPU processes. Through the experiment results, we can figure out the conclusions as follows:

(1) Generally speaking, proposed HW-SW codesign could get the best performance compared with only software or hardware encryption. For best case, HW-SW codesign could get 53.20% bandwidth improvement compared with only hardware encryption, and 20.11% improvement compared with AES-NI. The contribution comes from aggregation strategy and adaptive scheduling.

(2) As we concluded in Figure 24, the proposed methodology could provide a possibility for a better trade-off between performance and CPU idle. It is possible to get the same network bandwidth/RPS with AES-NI but still have additional available CPU resource. The user could utilize the saved CPU for other important tasks or take full advantage of it for further performance improvement.

6.3.2. Cipher Suite: ECDHE-RSA-DES-CBC3-SHA. Since 3DES is not supported by AES-NI yet, and the contribution with software encryption is little for HW-SW codesign, we tested the end-to-end performance for 2 working flows in this section. The first one is software encryption without AES-NI. The other one is hardware encryption with crypto engines. The RPS results for cipher suite ECDHE-RSA-DES-CBC3-SHA are shown as Table 8. We can get 1.6~2.1 times performance if we adopted hardware encryption.

We concluded the encryption bandwidth and the performance improvement in Figures 25 and 26. The maximum network bandwidth with hardware encryption could achieve 495.44 MB/S for large data blocks. For the same testing conditions, there are 60%~112% improvement compared with software encryption. Besides performance improved, there are still 13.62%~89.54% CPU idle available for hardware encryption.

According to the test in OpenSSL, there are maximum 12 times performance improvement. However, there are only 2 times RPS improvement for end-to-end testing. The problem is the constraint of the testing platform. We found the decryption speed with software in the client has been a bottleneck. All the CPUs are completely loaded for the client.

7. Conclusions and Future Work

In this work, we presented ACSA, Adaptive Crypto System based on Accelerators, which is able to adopt crypto mode adaptively and dynamically according to the request character and system load. We surveyed and analyzed different working flows with SSL/TLS firstly and found neither hardware nor software encryption can claim the best performance for all the application scenarios. Even there is advanced instruction acceleration for AES, the CPU

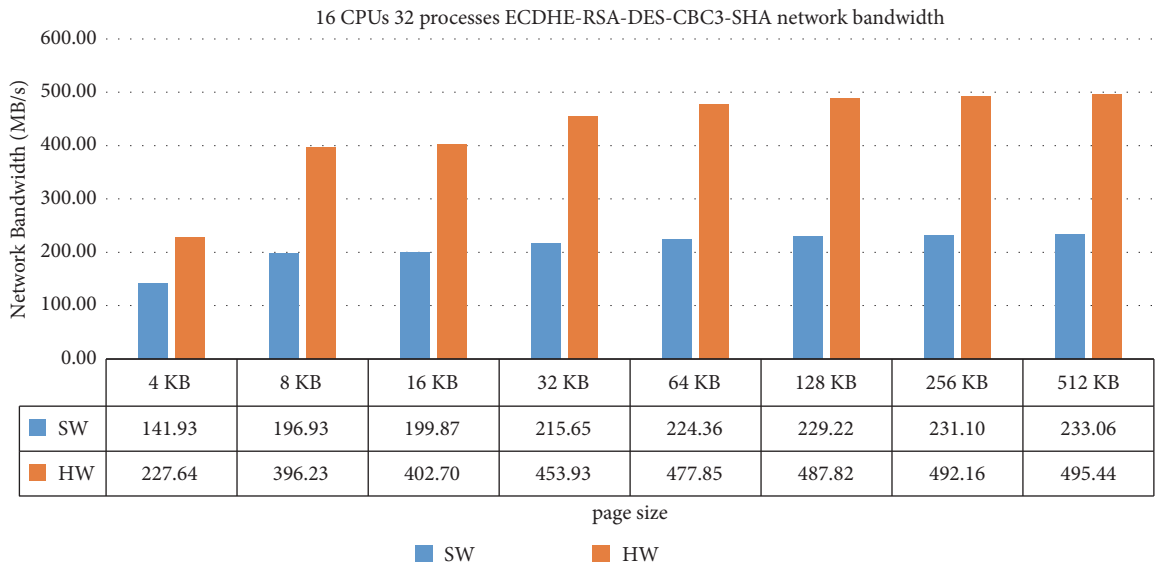


Figure 25: 16 CPUs 32 processes ECDHE-RSA-DES-CBC3-SHA network bandwidth.

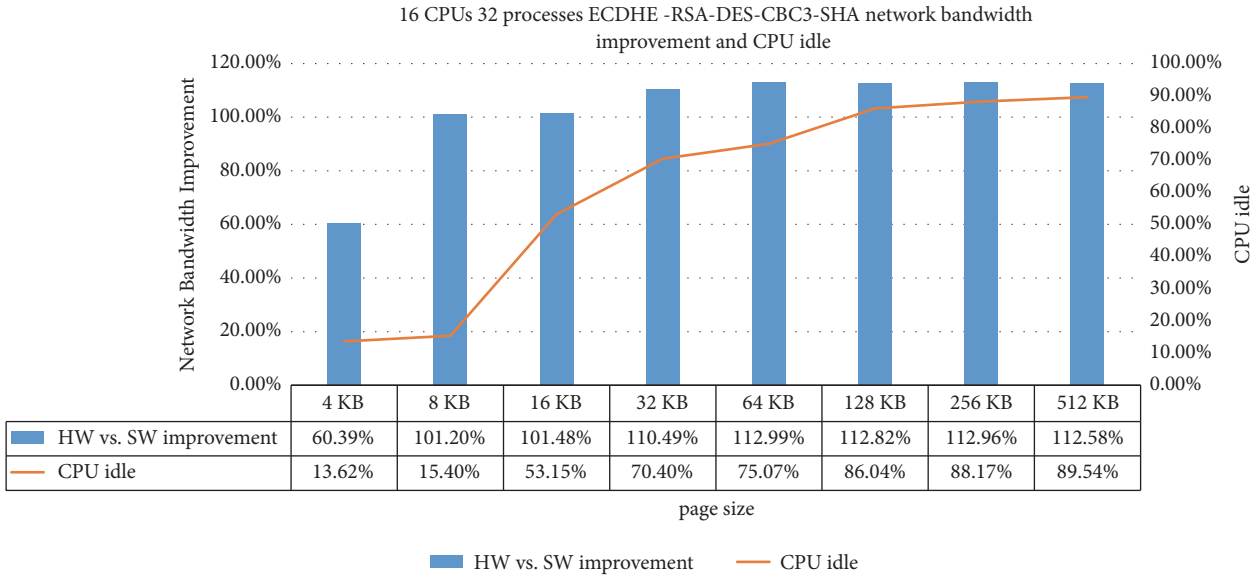


Figure 26: 16 CPUs 32 processes ECDHE-RSA-DES-CBC3-SHA network bandwidth improvement and CPU idle.

occupation is considerable compared with hardware crypto. Although HACs performs well for calculation, the invocation cost cannot be ignored for small data blocks. We not only proposed optimization strategies such as data aggregation to advance the contribution with hardware crypto engines, but also presented MM strategy (maximizing utilization with minimal overhead) and adaptive scheduling, to take full advantage of both software and hardware encryption. Through the establishment of 40 Gbps networking, we are able to evaluate the system performance in real applications with a high workload on various benchmarks and system configurations. For the encryption algorithm 3DES, which is not supported in AES-NI, we could get about 12 times acceleration with accelerators. For typical encryption AES supported by instruction acceleration, we could get 53.20% bandwidth improvement compared with only hardware encryption and 20.11% improvement compared with AES-NI. Furthermore, user could adjust the trade-off between CPU occupation and encryption performance through MM strategy, to free CPUs according to the working requirements.

Proposed design methodology possesses universal properties. The design flows of ACSA and MM algorithm are applicable to other similar designs with hardware acceleration for security web access [33]. As long as the design is heterogeneous architecture with CPUs and accelerators, the proposed design methodology is applicable regardless of the type of CPU and accelerator. Besides, this work is based on ARM server, which can be furthered for energy

efficiency exploration [34], providing emerging solutions for data center besides X86 based architectures [35, 36]. In future work, based on our current understanding of hardware and software encryption features, our research attempt is to study resource allocation strategies for heterogeneous server centers in different architectures from the perspective of energy efficiency rather than performance.

Data Availability

The data and codes are available on the lab server. Anyone that would like to obtain access could send an email to the corresponding author.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (61502061), Chongqing Application Foundation and Research in Cutting-Edge Technologies (cstc2015jcyjA40016), and the Fundamental Research Funds for the Central Universities (106112017CDJXY180004).

References

[1] N. Khan, I. Yaqoob, I. A. T. Hashem et al., "Big data: survey, technologies, opportunities, and challenges," *The Scientific World Journal*, vol. 2014, Article ID 712826, 18 pages, 2014.

[2] L. Li, K. Ota, Z. Zhang, and Y. Liu, "Security and privacy protection of social networks in big data era," *Mathematical Problems in Engineering*, vol. 2018, Article ID 6872587, 2 pages, 2018.

[3] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 1–11, 2011.

[4] J. Wilson, R. S. Wahby, H. Corrigan-Gibbs, D. Boneh, P. Levis, and K. Winstein, "Trust but verify: auditing the secure internet of things," in *Proceedings of the 15th ACM International Conference on Mobile Systems, Applications, and Services, MobiSys*, pp. 464–474, USA, 2017.

[5] A. Eric Young, J. Tim Hudson, and R. Engelschall, *OpenSSL: The Open Source Toolkit for ssl/tls*, 2011.

[6] S. Baskaran and P. Rajalakshmi, "Hardware-software co-design of AES on FPGA," in *Proceedings of the 2012 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2012*, pp. 1118–1122, India, August 2012.

[7] L. Bossuet, M. Grand, L. Gaspar, V. Fischer, and G. Gogniat, "Architectures of flexible symmetric key crypto engines-a survey: From hardware coprocessor to multi-crypto-processor system on chip," *ACM Computing Surveys*, vol. 45, no. 4, 2013.

[8] C. Maharak and B. Sowanwanichakul, "Security methods for web-based applications on embedded system," in *Proceedings of the IEEE TENCON 2004 - 2004 IEEE Region 10 Conference: Analog and Digital Techniques in Electrical Engineering*, pp. C56–C59, Thailand, November 2004.

[9] A. B. Smith, C. D. Jones, and E. F. Roberts, "Article David Brumley and Dan Boneh, Remote Timing Attacks are Practical," in *Proceedings of the 12th Usenix Security Symposium*, 2003.

[10] V. P. Nambiar and M. M. Zabidi, *Accelerating the AES Encryption Function in OpenSSL for Embedded Systems*, Inderscience Publishers, 2009.

[11] M. Khalil, M. Nazrin, and Y. W. Hau, "Implementation of SHA-2 hash function for a digital signature System-on-Chip in FPGA," in *Proceedings of the 2008 International Conference on Electronic Design, ICED 2008*, Malaysia, December 2008.

[12] D. B. Roy, S. Agrawal, C. Reberio, and D. Mukhopadhyay, "Accelerating OpenSSL's ECC with low cost reconfigurable hardware," in *Proceedings of the 2016 International Symposium on Integrated Circuits, ISIC 2016*, Singapore, December 2016.

[13] A. Thiruneelakandan and T. Thirumurugan, "An approach towards improved cyber security by hardware acceleration of OpenSSL cryptographic functions," in *Proceedings of the 1st International Conference on Electronics Communication and Computing Technologies 2011, ICECCT'11*, pp. 13–16, India, September 2011.

[14] C. Su, C. Wang, K. Cheng, C. Huang, and C. Wu, "Design and test of a scalable security processor," in *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference*, p. 372, Shanghai, China, January 2005.

[15] T. Isobe, S. Tsutsumi, K. Seto, K. Aoshima, and K. Kariya, "10Gbps implementation of TLS/SSL accelerator on FPGA," in *Proceedings of the 2010 IEEE 18th International Workshop on Quality of Service, IWQoS 2010*, IEEE, China, June 2010.

[16] H. Wang, G. Bai, and C. H. Zodiac, "System architecture implementation for a high-performance Network Security Processor," in *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, pp. 91–96, IEEE, Belgium, July 2008.

[17] R. Jeffrey, "Intel advanced encryption standard instructions (aes-ni)," Tech. Rep., Intel, 2010.

[18] M. Khalil-Hani, V. P. Nambiar, and M. N. Marsono, "Hardware acceleration of OpenSSL cryptographic functions for high-performance internet security," in *Proceedings of the UKSim/AMSS 1st International Conference on Intelligent Systems, Modelling and Simulation, ISMS 2010*, pp. 374–379, UK, January 2010.

[19] B. P. Kumar, P. Ezhumalai, and P. Ramesh, "Improving the performance of a scalable encryption algorithm (SEA) using FPGA," *International Journal of Computer Science and Network Security*, vol. 10, no. 2, 2010.

[20] D. Jacquet, F. Hasbani, P. Flatresse et al., "A 3 GHz dual core processor ARM cortex TM -A9 in 28 nm UTBB FD-SOI CMOS with ultra-wide voltage range and energy efficiency optimization," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 4, pp. 812–826, 2014.

[21] Z. Ou, B. Pang, Y. Deng, J. K. Nurminen, A. Ylä-Jääski, and P. Hui, "Energy- and cost-efficiency analysis of ARM-based clusters," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012*, pp. 115–123, Canada, May 2012.

[22] J. L. Bez, E. E. Bernart, F. F. dos Santos, L. M. Schnorr, and P. O. Navaux, "Performance and energy efficiency analysis of HPC physics simulation applications in a cluster of ARM processors," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 22, p. e4014, 2017.

[23] C. D. Schmidt and C. D. Cranor, *Half-Sync/Half-Async*, 1998.

[24] X. Zhang and K. K. Parhi, "High-speed VLSI architectures for the AES algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 9, pp. 957–967, 2004.

[25] P. Chodowiec and K. Gaj, "Very compact FPGA implementation of the AES algorithm," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, Heidelberg, Berlin, Germany, 2003.

[26] G. Singh, "A study of encryption algorithms (RSA, DES, 3DES and AES) for information security," *International Journal of Computer Applications*, vol. 67, no. 19, pp. 33–38, 2013.

[27] G. Piyush and K. Sandeep, "A comparative analysis of SHA and MD5 algorithm," *Architecture 1*, p. 5, 2014.

[28] J. Viega, P. Chandra, and M. Messier, *Network Security with OpenSSL*, Oreilly Publications, 1st edition, 2002.

[29] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra, and R. Ross, "Improving I/O forwarding throughput with data compression," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing, CLUSTER 2011*, pp. 438–445, USA, September 2011.

[30] A. Timor, A. Mendelson, Y. Birk, and N. Suri, "Using underutilized CPU resources to enhance its reliability," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 94–109, 2010.

[31] http://nginx.org/en/.

[32] http://httpd.apache.org/docs/current/programs/ab.html/.

[33] http://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html.

[34] D. Kline, N. Parshook, X. Ge et al., "Holistically evaluating the environmental impacts in modern computing systems," in *Proceedings of the 7th International Green and Sustainable Computing Conference, IGSC 2016*, pp. 1–8, China, November 2016.

[35] K. Jonathan, "Growth in data center electricity use 2005 to 2010," *A report by Analytical Press, completed at the request of The New York Times 9*, 2011.

[36] A. K. Jones, "Green computing: new challenges and opportunities," in *Proceedings of the on Great Lakes Symposium on VLSI*, p. 3, ACM, 2017.