HEF: A Hardware-Assisted Security Evaluation Framework

by

Sukwha Kyung

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved May 2017 by the
Graduate Supervisory Committee:

Gail-Joon Ahn, Chair
Adam Doupé
Ziming Zhao

ARIZONA STATE UNIVERSITY

December 2017

ABSTRACT

Hardware-Assisted Security (HAS) is an emerging technology that addresses the shortcomings of software-based virtualized environment. There are two major weaknesses of software-based virtualization that HAS attempts to address – performance overhead and security issues. Performance overhead caused by software-based virtualization is due to the use of additional software layer (i.e., hypervisor). Since the performance is highly related to efficiency of processing data and providing services, reducing performance overhead is one of the major concerns in data centers and enterprise networks. Software-based virtualization also imposes additional security issues in the virtualized environments. To resolve those issues, HAS is developed to offload security functions from application layer to a dedicated hardware, thereby achieving almost bare-metal performance and enhanced security. As a result, HAS gained more popularity and the number of studies regarding efficiency of the technology is increasing.

However, there exists no attempt to our knowledge that provides a generic test mechanism that is universally applicable to all HAS devices. Preparing such a testbed for each specific HAS device is a time-consuming and costly task for hardware manufacturers and network administrators. Therefore, we try to address the demands of hardware vendors and researchers for a generic testbed that can evaluate both performance and security functions of the HAS-enabled systems.

In this thesis, the HAS device evaluation framework (*HEF*) is defined for hardware vendors, network administrators, and researchers to measure performance of the system with HAS devices. *HEF* provides a generic test environments for a given HAS device by providing generic test metrics and evaluation mechanisms. *HEF* is also designed to take user-defined test metrics and test cases to support various hardware. The framework performs the entire process in an automated fashion, and thus

it requires no user intervention. Finally, the efficacy of *HEF* is demonstrated by performing a case study using Intel QuickAssist Technology (QAT) adapter, which is a dedicated PCI express device for cryptographic tasks.

DEDICATION

To God, the one gives me the strength to stand up again whenever I'm about to fall down and tempted,

To my parents, for their unconditional love, support, and sacrifice,

To my brother, for always being there and making me to do my best.

ACKNOWLEDGMENTS

This thesis received a lot of help and support from other people. I would like to thank some of them:

First of all, I would like to express my special gratitude to my thesis advisor and committee chair, Dr. Gail-Joon Ahn for his motivation, guidance and support. This thesis would not have been possible without his brilliant idea and pointers to guide me throughout this project. I could not have imagined a better mentor for my study.

I would like to thank Dr. Adam Doupé, for being a part of the committee, for all his helps and support. His insightful comments helped me refine this thesis.

My sincere thanks also goes to Dr. Ziming Zhao, another committee memeber of this thiesis, for his helps, patience and support. His hard questions made me constantly perform research and move forward.

Last but not least, I thank my fellow labmates in SEFCOM:
Dr. Wonkyu Han, for the days and nights we worked together. I am grateful for his mentoring and motivation that led me to the first glance of research.
Naveen Tiwari, for all the fun we had while we were working together, and for his comments, questions and constant help.
Vaibhav Dixit, Haehyun Cho, Dr. Carlos Rubio, Michael Mabey, and Erik Trickel, for their help and support.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Advance in hardware technology during the past few decades led to an emergence of new classes of computers. Mobile devices (cell phones, tablet computers, etc), for example, have become one of the primary computing platforms these days. Computing capability of those devices also outperforms that of the desktop computers from merely a decade ago. As the processing power increases, the amount of data that are processed by servers in data centers have also drastically increased. Today, data centers around the world process incredible amount of data that is exchanged via communication channels and networks. It is not only the hardware that advanced over time — technologies for utilizing the ample resources of high-performance systems in data centers has also advanced.

Virtualization is one of such technologies that has emerged not only in enterprise networks but also in other systems including personal and mobile platform. The main purpose of virtualization is to create a virtual platform (whether it is in a form of network interface, storage, or software component) to share the ample, physical resources of the host system. In this way, multiple platforms or services can be provided with less cost and the waste of computing resources can also be prevented.

Nevertheless, performance is still a major concern in today's data centers. This is mainly because virtualized environments impose significant performance overhead to the host. Such performance overhead is mainly caused by additional software layer called hypervisor. Hypervisor manages execution of guest systems and sharing of host resources among those guest environments, and the functions of hypervisor as a mediator between host and guest systems create the overhead. There are two types

of hypervisors — Type-I and Type-II hypervisors. While Type-II hypervisor is the most popular type of hypervisor in today's virtualization technology, it is also main source of the performance overhead (see Chapter 2).

In addition to the performance overhead caused by hypervisor, the increasing amount of data exchanged throughout the network also affects the performance of servers in data centers. For example, annual global IP traffic forecast published by CISCO anticipates the global IP traffic will increase threefold up to 2020 [6]. This phenomenon is natural because the number of services provided by today's network is increasing (for e.g., images, video streams, cloud database, etc). Thus, the fact that amount of data is increasing makes the performance a critical factor that must be considered by network administrators.

Another important aspect that must be considered in the virtualized computing environments is security, i.e., how to provide secure execution environment for the virtualized environments. Other than the fact that techniques and tools used by attackers constantly advance, virtualization itself imposes numerous security issues [26],[22]. Therefore, hardening the security of those environments becomes critical for service providers. Consequently, the architecture in data centers and high performance computing sites is in constant state of change with cutting-edge technologies to balance hardware cost, operational expenditures, performance and security.

Hardware-Assisted Security (HAS) has emerged in an attempt to address the aforementioned disadvantages of virtualization technology. HAS device offloads security functions from software layer to a dedicated hardware, and thus, increasing performance and providing hardened security to the system. In fact, HAS is already a popular choice in embedded systems architecture and is also being applied to enterprise data centers and high performance computing setups to achieve almost bare-metal performance. Therefore, it is of a great importance for hardware manu-

facturers, system administrators, and researchers that they are able to validate and analyze the performance of HAS products. However, most studies related to HAS or hardware-assisted virtualization environments in high performing platforms and data centers concentrate only on either benchmarking the performance of a specific HAS device or enhancing security through HAS with minimum performance overhead [5],[25],[24].

In this thesis, a design for an automated, generic framework for evaluating performance of HAS devices is proposed. The HAS hardware evaluation framework (named *HEF*) defined in this thesis enables hardware vendors, system administrators, and researchers to analyze performance of the system with a given HAS device. To this end, *HEF* provides a generic performance analysis mechanism that can be applied to any HAS hardware in standard servers. *HEF* is also designed to take user-defined test metrics and test-cases to support various test cases. *HEF* requires no user intervention after taking user inputs, and thus, provides an enhanced usability through automation. The efficacy of *HEF* is demonstrated by performing a case study using Intel QuickAssist Technology (QAT) adapter, which is a dedicated PCI express (PCIe) device for various cryptographic tasks.

The contributions made by this thesis are summarized as follow:

1. The design for a generic, automated performance analysis framework for evaluating HAS hardware, named *HEF*, is proposed.

2. *HEF* provides an environment for hardware vendors to validate the security functions of given HAS device. It also performs performance benchmark to measure the performance of the system on which the HAS device is installed. Then, *HEF* generates a comparative analysis using the performance benchmark

results from HAS-enabled and HAS-disabled environments.

3. An instance of *HEF* is implemented to demonstrate its efficacy and effectiveness. For case study, Intel QAT adapter, one of the HAS devices, is used in this thesis for the demonstration.

Structure of the remaining thesis is as follow:

The motivation of this thesis are listed in Chapter 3. After the goals are clearly identified, the detailed architecture of *HEF* are discussed in Chapter 4, along with the explanation of how the framework is implemented in Chapter 5. The functionalities of *HEF* is demonstrated using Intel QAT device and the results are analyzed in Chapter 6. And then, the efficacy and limitations of the framework are discussed in Chapter 9. Other works that influenced *HEF* are introduced in Chapter 8. Finally, Chapter 10 concludes this thesis.

Chapter 2

BACKGROUND ON HARDWARE-ASSISTED SECURITY

In this chapter, we introduce the background knowledge required to understand what HAS is. In order to understand the history and mechanism of HAS, we first have to look at virtualization; its advantages and disadvantages, and attempts to address those disadvantages.

2.1 Virtualization

In computer architecture domain, virtualization means an insertion of a layer in the logical stack to abstract underlying hardware or software layer. More specifically, virtual machines represent computing environments that simulate and share the existing hardware resources of the host system. One of the main reasons for using the technology is to avoid wasting of processing power and improve resource utilization by abstracting the host resources. Virtualization is also used to address specific problems, such as providing support for legacy functionality, and standardizing interface in logical models [16].

Advance in hardware and cloud computing technologies make virtualization technology a popular solution that can provide on-demand, elastic, and highly efficient self-service in terms of resource utilization [14]. In particular, primary benefit of virtualization for enterprise networks and data centers is that it reduces the waste of computing resources. The technology also improves stability and reliability of the system because any errors, or even crash, taking place in guest systems does not affect the host.

In virtualization, the guest systems are running on top of a piece of software

called *hypervisor*, which is the core software layer in virtualization technology [20]. At high level, hypervisor enables the guest machines to share computing resources of the host machine. To enable the guest systems to share the host resources and communicate with the underlying hardware efficiently, hypervisor simulates machine I/O instructuctions. There are two types of hypervisor; one is Type-I hypervisor (or bare-metal hypervisor), which is running directly on hardware. Since Type-I hypervisor is running without host operating system, it minimizes performance overhead by communicating with the underlying hardware directly. For the same reason, Type-I hypervisor is also considered to be more secure than Type-II hypervisor. A virtualized environment using Type-I is shown in Figure 2.1. However, the cost for maintenance and updating hardwares with bare-metal hypervisor is high, because the hypervisors are running as an embedded software on hardware. It also requires certain skill sets for operation.

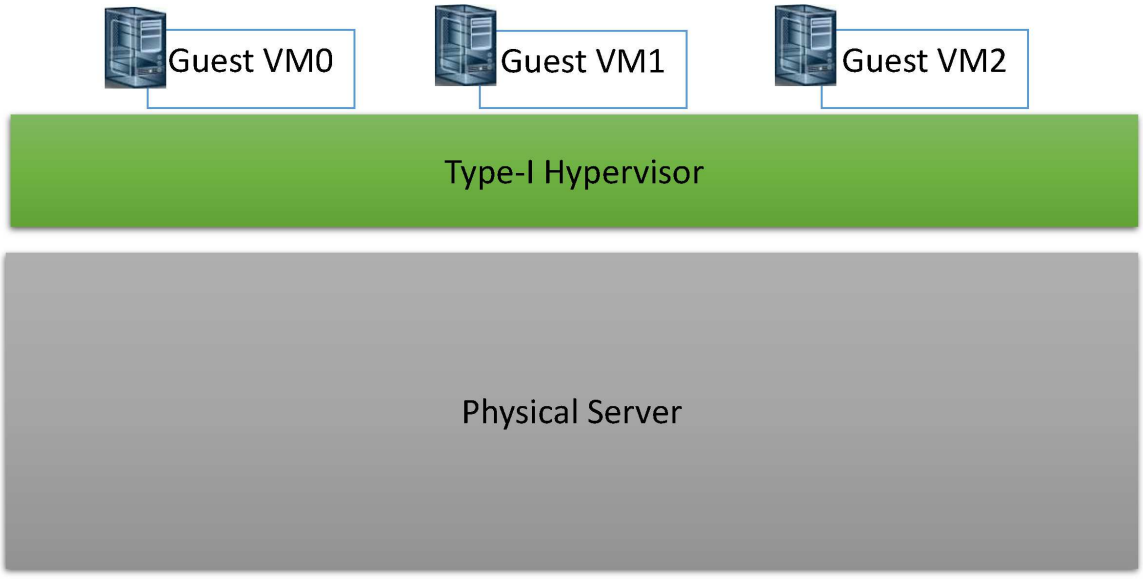

Figure 2.1: Virtualized Environment using Type-I Hypervisor

The other type of hypervisor is Type-II hypervisor (or hosted hypervisor), which runs on top of host operating system. Since the host operating system manages

hardware, the hosted hypervisor can support various hardwares. The hosted hypervisors are also easy to setup and the cost is low. Hosted hypervisors, however, cause higher performance overhead and low stability and security due to more points of failure compared to bare-metal hypervisors. Figure 2.2 illustrates a virtualization using Type-II hypervisor.

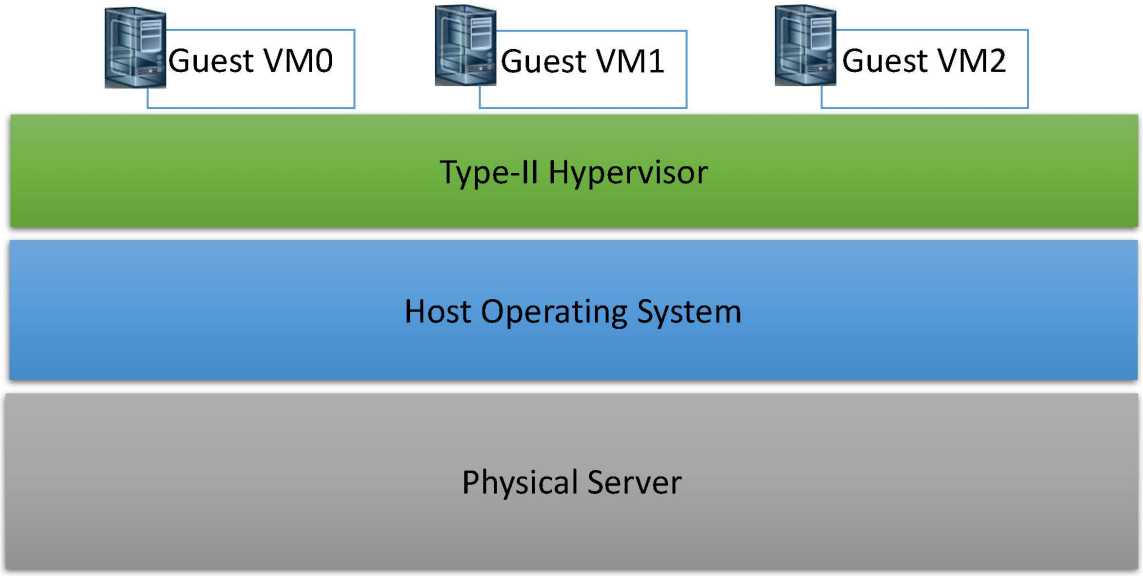

Figure 2.2: Virtualized Environment using Type-II Hypervisor

Modern data centers and middle-to-large enterprise networks use virtualization as the technology that enables host systems to share their ample resources. In other words, each server can dynamically provision multiple services to users through the technology. The trend in virtualization was utilization of bare-metal hypervisors because of their advantages. However, traditional virtualizion technology, regardless of the type of hypervisor used, imposes two major challenges. One is performance overhead. Even with today's advanced hardware, virtualization creates significant performance overhead as the hypervisor is running as a piece of software intercepting all interaction between the guest systems and host hardware [1],[21]. Consequently, there could be 100 – 400% performance overhead caused by hypervisors [15], es-

pecially when Type-II hypervisors are used. There is another type of virtualization technology that attempts to minimize performance degradation — paravirtualization. paravirtualization [2] modifies the guest systems to make system calls to the hypervisor rather than emulating the hardware environment. In this way, the workload of emulating hardware for hypervisor decreases, and hence decrease the performance overhead. However, it does not totally eliminate the overhead because the hypervisors still act as a mediator between host hardware and guest systems. The actions as a mediator include creating device driver for guest systems, sending instructions from guest to host, and accessing the host hardware on behalf of the guest systems.

The other challenge is security issues related to the technology. Additional software layer created by using hypervisors introduced new vulnerabilities and attacks. For example, VM escaping, VM sprawling, and attack between VMs are among the new threats [13] imposed by virtualization. Those new vulnerabilities obviously enable attackers to compromise both the guests and host systems if they are not managed properly. In other words, hypervisors introduces multiple attack surfaces and points of failures. Fortunately, there are numerous studies that attempt to address the hypervisor-related issues [13],[23],[19],[18]. However, those studies still fail to address the performance overhead issue, or even impose more overhead as the solution utilizes extra software in an attempt to resolve the given problem [3].

2.1.1 Hardware-Assisted Virtualization and Hardware-Assisted Security

Performance and security issues remain for any type of virtualization technologies as long as additional software layer (i.e., hypervisor) exists. HAS is an emerging technology that attempts to address the issues of virtualization. HAS is, in essence, utilizes hardware-assisted virtualization. Unlike other types of virtualization that uses hypervisors, HAS hardware achieves full access to physical function (PF) of

host hardware without intervention of hypervisors. The core hardware-assisted virtualization technology leveraged by HAS hardware is called single-root input/output virtualization (SR-IOV).

SR-IOV is a standard for I/O device virtualization. Any I/O device with SR-IOV (whether it is HAS device or not) creates multiple virtual functions (VFs) that are shared by multiple guest virtual machines [7]. In addition, SR-IOV creates direct I/O path between the device and guest virtual machine, instead of emulating I/O instructions for PF. In this way, bare-metal performance is achieved by having a dedicated hardware for certain security-related tasks and minimizing the intervention of hypervisors [7]. Figure 2.3 illustrates a virtualized environment using SR-IOV enabled device. Majority of security functions provided by today's HAS devices is various cipher suites for encryption/decryption of data that are either at rest (i.e., data stored in a database) or in transit.

Security-related issues in traditional virtualization can also be resolved by using specific security VFs provided by HAS devices. This is because those security functions provided by a certain hardware guarantees more security and trustworthiness than software-based security functions, which are more vulnerable to modification and reverse engineering [27].

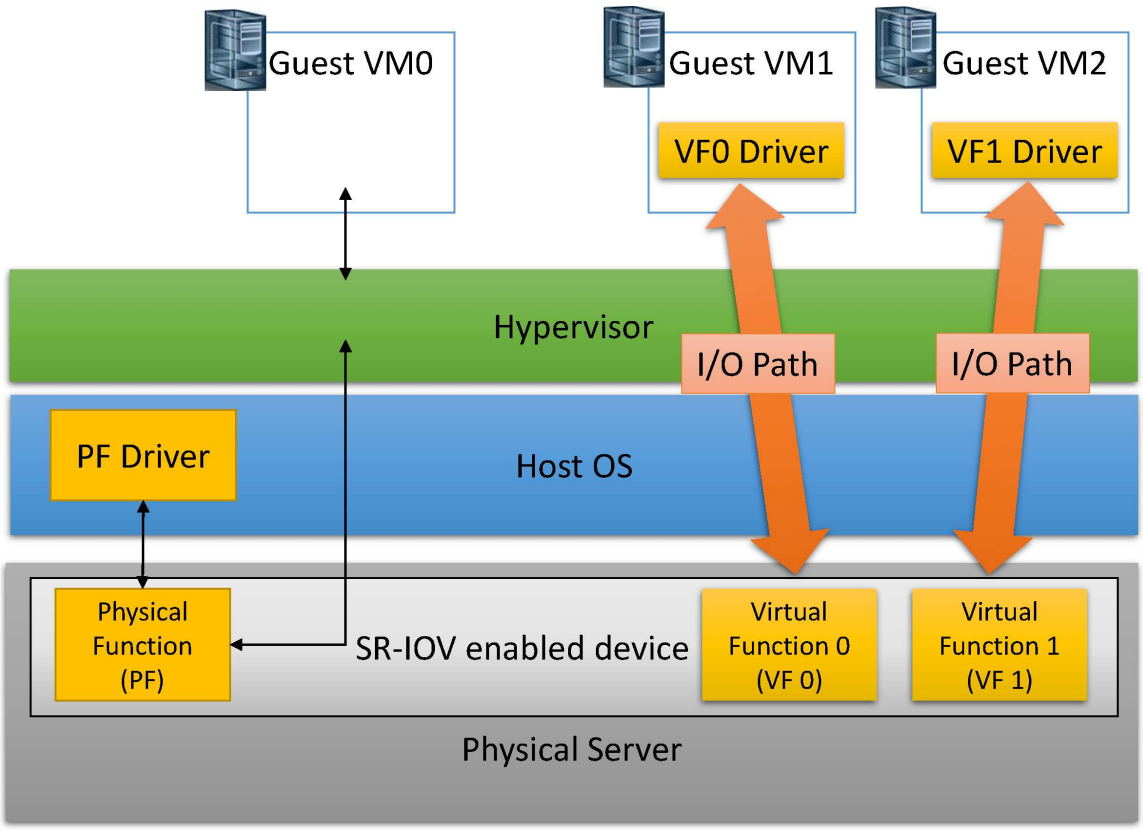

Figure 2.3: Virtualized Environment with SR-IOV

Chapter 3

PROBLEM STATEMENT

In this Chapter, we define the problems we want to address first. Then, based on the problems identified, we also define the goal of this thesis. In general, we aim to design and implement a fully automated, generic performance analysis framework that can be applicable to any HAS devices. The problems we identified and specific goals to address those problems are as follow:

1. With the rise of HAS techonlogy, there has been a lot of studies regarding its application in data centers and high performance computing setups. However, there is no attempt to provide an universal performance analysis mechanism for HAS devices. To our best knowledge, existing frameworks are in ad-hoc manner — in other words, those frameworks are device-specific. *HEF* addresses this issue by not only performing a performance benchmark but also providing a comparative analysis of using HAS technology.

2. Current studies only concentrate on either enhancing the security of system using HAS technology or increasing the performance by using hardware virtualization. A goal of this thesis is to encompass both aspects by demonstrating the security functions of the system with a HAS device *and* analyze performance of the system.

3. Other existing benchmarking frameworks are designed in ad-hoc manner and only for their target HAS devices. Another goal of this thesis is to devise a *generic* framework that is applicable to all HAS devices in standard servers.

4. Those existing frameworks also require a lot of time, skills, and costs to create and run the benchmark. *HEF* automates the benchmark process so that it can also minimize the effort and enhance usability.

Chapter 4

FRAMEWORK DESIGN AND ARCHITECTURE

The design requirements of the framework is discussed in this chapter, along with the details of architecture and components of the framework. We also define input and output of *HEF* in this chapter.

4.1 Design Goals

First, we define the design goals of *HEF*. The limitations of other existing frameworks for HAS devices include their limited applicability to other devices, high cost for setup, and low usability. The framework devised in this thesis attempts to address those issues by achieving the following design goals:

1. **Automation:** The framework must be able to perform the benchmark without user intervention. In other words, *HEF* runs the entire performance benchmark and analysis by itself from the moment the user provides input. *HEF* should also be able to generate the output and provide it to the user. Thus, the entire process from test case generation to results analysis should be automated.

2. **Generalization:** The framework should be applicable to any HAS device unless it requires specific set of hardware. One of the assumptions in this thesis is that target HAS devices are installed in standard servers (see Chapter 9). To be applicable to any HAS devices, the framework can also take in user-defined inputs for test metrics. In this way, *HEF* can support various test cases. Thus, the framework defined in this thesis should be able to perform both the security function demonstration and performance analysis for any HAS devices that are

used in standard servers.

3. **High Modularity:** To achieve high usability and minimize the effort, *HEF* consists of several modules, each of which performing certain automated task required for performance benchmark of the given HAS device. High modularity also enables users to plug in the test cases that they generated easily. It is also easy for them to troubleshoot.

4.2 Architecture

This section presents the overall architecture of the framework, followed by descriptions for each of its components. At high level, the framework consists of three agents, and some of each agent is also composed of a group of sub-modules. Each agent represents a specific task during the performance analysis process. The whole architecture of the framework is illustrated in Figure 4.1. The specific descriptions for each component is as follow:

1. **Preparation Agent:** This agent is responsible for processing inputs, generating test-cases, and configuring test environments. Preparation agent consists of the following sub-modules for specific tasks:

   Input Collection Module: This agent is responsible for managing default and user-defined input. Each input (i.e., test metrics) is managed by two sub-modules (Default Input Manager and User-defined Input Manager) in the Input Collection Agent. Default Input Manager and User-defined Input Manager simply take the inputs through the user interface, parse the input, store, and give the processed input to the other modules when necessary. User-defined Input Manager can also take other user-defined specifications, such as different

test metrics, test cases, and software tools. In this way, *HEF* can support different test cases for various HAS devices.

There are two types of inputs used in *HEF*. Default inputs are the ones that are given to the user by default. Default inputs (i.e., default test metrics, test cases and software tools) are selected as defined in RFC 2544, which is a standard for benchmarking communication devices [4]. User-defined input is the other type of input that can be plugged into User-defined Input Manager. This agent is also responsible for retrieving resource information from the system.

Test-case Generation Module: As the name indicates, Test-case Generation Agent generates test cases that for performance benchmark using the inputs. The other responsibility of this module is storing and retrieving the user-defined test cases when necessary.

Test Environment Configuration Agent: Based on the test-cases generated, Test Environment Configuration Agent configures the testbed and creates test environment using the list of software tools (whether default or user-defined). The list of software tools that are defined by the user is passed to this agent and used to download the required software during the preparation of the test environment.

2. **Test Agent:** Test Agent is one of the two key components of *HEF*. This agent collects processed inputs and test-cases from appropriate agents and runs the benchmark using them. After collecting the benchmark results, Test Agent stores them in the result database. This agent is also responsible for error reporting by logging any exceptions/errors and passing them to the user for troubleshooting.

3. **Analysis Agent:** This agent is the other key component of *HEF*. It retrieves

raw result data from the result database and performs a comparative analysis to visualize performance enhancement by using the HAS device. The main responsibilities of this agent is, therefore, performing analysis and visualizing the analysis results.

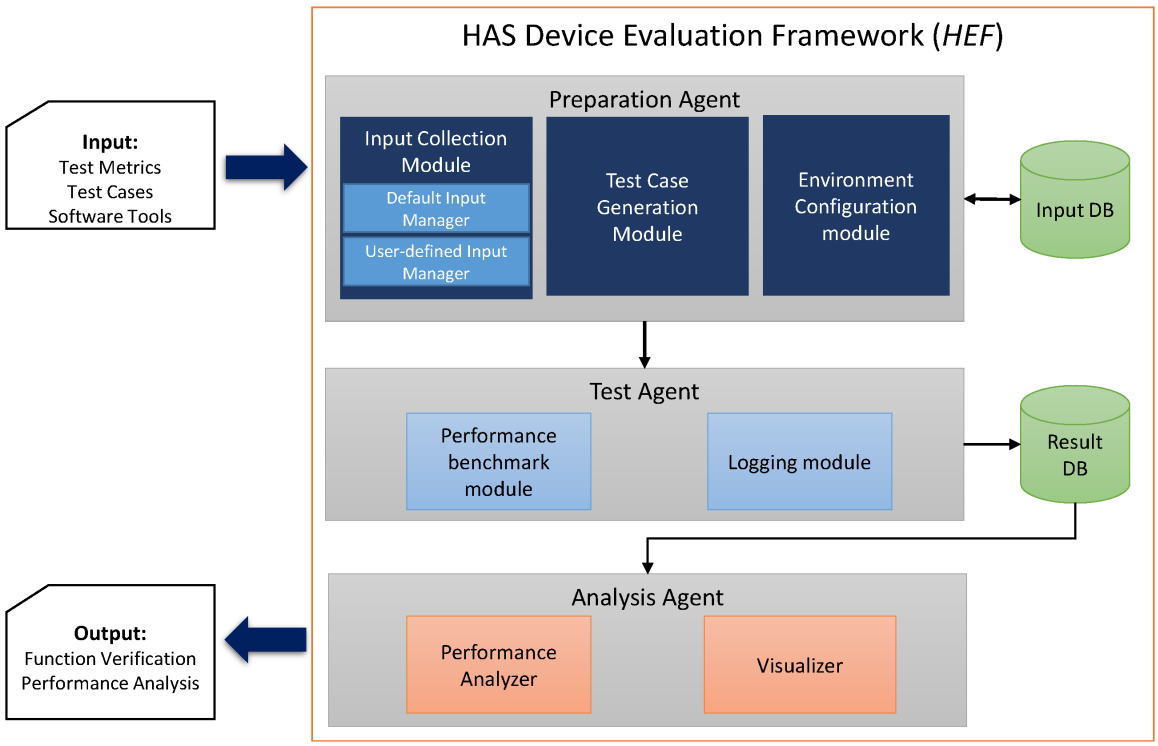

Figure 4.1: Architecture Overview of *HEF*

4.2.1 Input Definition

The input types for the framework are also defined in this section because *HEF* takes not only one type of inputs but a series of multiple types of inputs. The framework allows the user to choose either one or both types of input, which is used by the framework to generate test cases and run the benchmark. Table 4.1 lists possible series of input. First, all types input fall into on of the following categories:

Input Type	Categories
Default Input	Test Metrics, Test Cases, Software Tools
User-Defined Input	

Table 4.1: Types and Categories of Input

1. **Test Metrics:** The default test metrics are defined according to RFC 2544 [4], which is the benchmarking methodology for network interconnect devices defined by Network Working Group. It includes three test metrics — *Bult Throughput* (Gbps), *Connections per second* (CPS), and *CPU Utilization* (utilization per core measured in %). The user can define test metrics and other categories of the input, so that the framework can perform the benchmark as intended.

2. **Test Cases:** Using the test metrics defined, the framework generates test cases for the benchmark. This can be also defined by the user, and therefore, the user-defined test cases can be an input in this case. In this thesis, the default test cases are generated by using Intel QuickAssist Device (QAT), which is used for the case study (see Chapter 6). The case study demonstrates the efficacy of *HEF*.

   A combination of different elements are used to create the default test-cases, dividing the test-cases into five categories:

   (a) Environments that defines two different configurations of the testbed — the web server-client and web server-proxy-client environments.

   (b) QAT option — either enabled or disabled

   (c) Cipher suite used for each test case

   (d) The number of CPU cores used on the device under test (DUT) device,

and

(e) The default test metrics

Categorizing the test cases using the elements above, over 480 cases are generated by the framework.

3. **Software Tools:** This category is created to allow the users to use specific software tools required to run his/her own test-cases. The default tools provided by the framework include *iperf* for the bulk throughput, and *top* and *mpstat* for the cpu utilization. CPS is measured by using web servers that are configured to measure the number of connections. More details about how the benchmark is performed are explained in Chapter 6.

The inputs for the framework are then divided into two main types:

1. **Default Input:** This type of inputs are provided to the users by default. The user can choose whether the given inputs are used or not in the benchmark.

2. **User-defined Input:** This is a type of input that can be defined by the users and plugged into the framework through the user interface. Due to the high modularity supported by the framework, the user can easily plug-in the customized input, test-cases, and required software tools to the framework. However, the user must define and provide all three items (i.e., test metrics, test cases, and software tools) correctly in order for the framework to perform the benchmark.

Chapter 5

IMPLEMENTATION

An instance of *HEF* is implemented to demonstrate its functionalities and efficacy. As discussed in Chapter 4, five different agents are implemented using Python 2.7.6, JavaScript, and HTTP in approximately $3,500$ lines of code, supporting the design goals explained in Section 4.1. More specifically, all five agents are implemented using Python langauge, while the web user interface is implemented using JavaScript and HTTP for enhanced usability, which is also one of the design goals.

For the later use of experimental results, the analysis agent in *HEF* is isolated from the testing agent and runs comparative analysis to effectively showcase performance gaps between QAT-installed and non-QAT settings. The raw experimental data are stored as spread sheets (.csv format) and graphs (.png format) after all the verification and benchmark is complete.

Chapter 6

CASE STUDY

In this section, we demonstrate the functionalities and efficacy of *HEF* using Intel QuickAssist Technology (QAT) device. The detailed explanation on experiment setups are provided. The output (i.e., the experimental results) is then discussed.

For the case study, we used Intel QAT. Intel QAT is a PCI Express (PCIe) hardware that accelerates and compresses cryptographic workloads by offloading the task to a dedicated hardware through SR-IOV [9]. In other word, one PF of a device can be divided into multiple VFs, as explained in Chapter 2. Security functions provided by QAT are various cryptographic functions, including symmetric cryptography (AES, DES,etc), elliptic curve cryptography (ECDSA and ECDH), and hash algorithms (SHA-1, SHA-2, etc). Those functions are run and tested in the web server-client and server-proxy-client environments (see Section 6.1.4). To perform the benchmark, test-cases for each cryptographic function are given to the framework, along with other elements required (i.e., test metrics and software tools required to configure the environment).

6.1 Test Cases and Environment

In this section, the methods used to create test cases and environments are discussed. The hardware and software specifications required for the physical testing servers are also explained. At high level, performance of two environments with and without QAT is measured. One test environment is a plain web server-client setup, while the other environment consists of a web server, client, and a proxy in-between the two.

6.1.1 Test Case Generation

To measure CPS and Bulk throughput, more than two different entities (server and client) must exchange the data so that the traffic between them can be monitored and analyzed. Also, other elements are considered in generating test-cases other than test metrics and cryptographic functions provided by the QAT. Those other criteria include configuration of test environments, the presence of QAT in the testing environment, number of CPU cores, and the maximum transmission unit (MTU). These criteria are selected to create various test-cases so that we can compare the performance of the device under testing (DUT) with different configurations.

Configuration of environments defines two different test environments — a typical web server-client and web server-proxy-client environment. The purpose of having a proxy in-between the server and client is to obtain the performance results in different network environments and compare them with the web server-client setup. Similarly, the number of CPU cores and MTU are selected to try out different configurations and compare the performance difference. Finally, impact of the presence of QAT (whether QAT is installed or not) is measured in terms of performance to prove efficiency of the HAS device. Table 6.1 lists all the criteria for generating test cases. With all possible combinations, over 450 test cases are generated. The test cases generated are given as inputs to *HEF* and the benchmark is performed automatically, without any user intervention.

6.1.2 Hardware Ingredients

Table 6.2 lists the hardware specifications used in this case study. It contains all the hardware components for both the DUT and client. All hardware components are carefully selected so it assembles standard servers used in today's enterprise networks.

A	**B**	**C**	**D**	**E**	**ETC**
Environment	*QAT option*	*Cipher*	*# of cores*	*Measurement*	*MTU*
Server-Client, Server-Proxy-Client	Non-QAT, QAT	AES128-SHA, AES128-GCM-SHA256, ECDHE-ECDSA-AES128-SHA, ECDHE-RSA-AES128-SHA	1, 2, 4, 6, 8	Bulk Throughput, CPS, Utilization	1500, 9000

Table 6.1: Test Cases Generated for Intel QAT Device

Category	**Description**	**Qty**
DUT CPU	Intel E5-2699v3	1
Client CPU	Intel E5-2658v3	1
Memory	Samsung 8GB 288-Pin DDR4 SDRAM ECC Registered DDR4 2133 (PC4 17000) 2RX8, Server Memory Model M393A1G43DB0-CPB0	8
QAT Adapter	Intel QuickAssist Adapter 8950-SCCP	2
NIC X520(2xPorts)	Intel Ethernet Converged Network Adapter X520-DA2 - Network adapter - PCI Express 2.0 x8 low profile	1
NIC X710(4xPorts)	Intel X710 Network Adapter 10GB PCIe-3.0 x8 SFP+ x 4 X710DA4FH	1

Table 6.2: Hardware Components for Physical Servers

6.1.3 Software Ingredients

Table 6.3 lists the software components used in the benchmark. It contains all the software components for both the DUT and client servers. We also modified the source code of NGINX web server and encryption library of OpenSSL to make them compatible with QAT and count the number of connections for CPS.

Category	Description
Operating System	Fedora 20
Kernel	Linux kernel 3.11.5
Hypervisor	Oracle VirtualBox 5.1.18
Software packages	C Development Tools and Libraries Development Tools kernel-devel zlib-devel glibc-devel openssl-devel sysstat
OpenSSL	AES-NI ver 1.0.2g
Web server	NGINX ver 1.6.2
CPU utilization measuring tool	mpstat
Remote script execution	parallel-ssh

Table 6.3: Software Components Required for Case Study

6.1.4 Test Environments

Figure 6.1 and 6.2 illustrates the testbed setup for the web server-client and web server-proxy-client environments respectively. To measure how much performance improvement QAT can achieve, the web server (i.e., the DUT) is implemented as a VM assigned with the VFs of QAT, while the traffic generator (client) is implemented in separate physical server. Similarly, the web server-proxy-client is created using two VMs for web server and proxy to simulate a simple virtual network. The configuration of those environments are performed by Environment Configuration module of *HEF*.

Those configurations performed by *HEF* include updating kernel to the compatible version with QAT, downloading and installing required drivers and software tools.

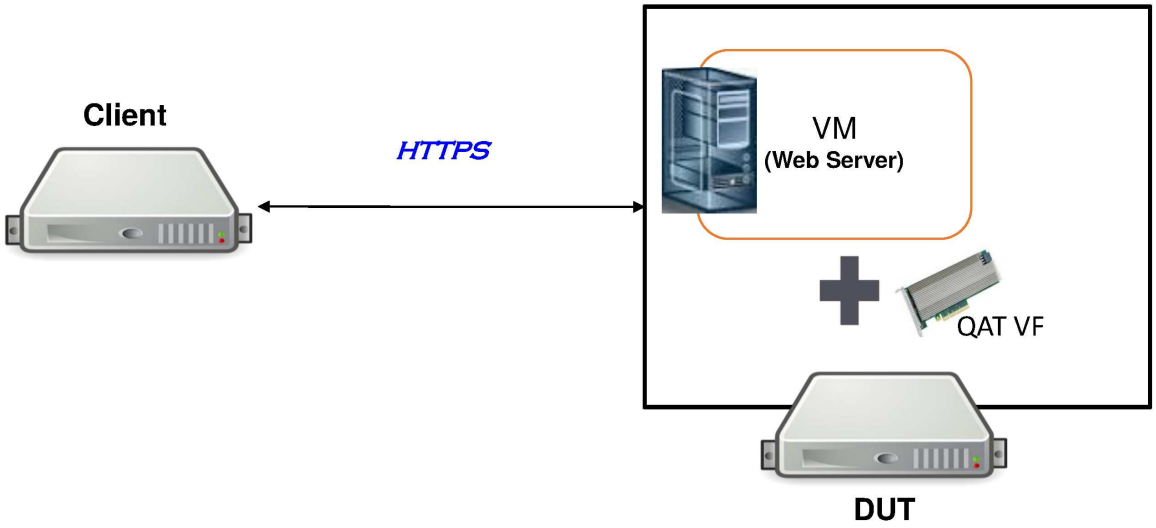

Figure 6.1: Web Server-Client Test Setup

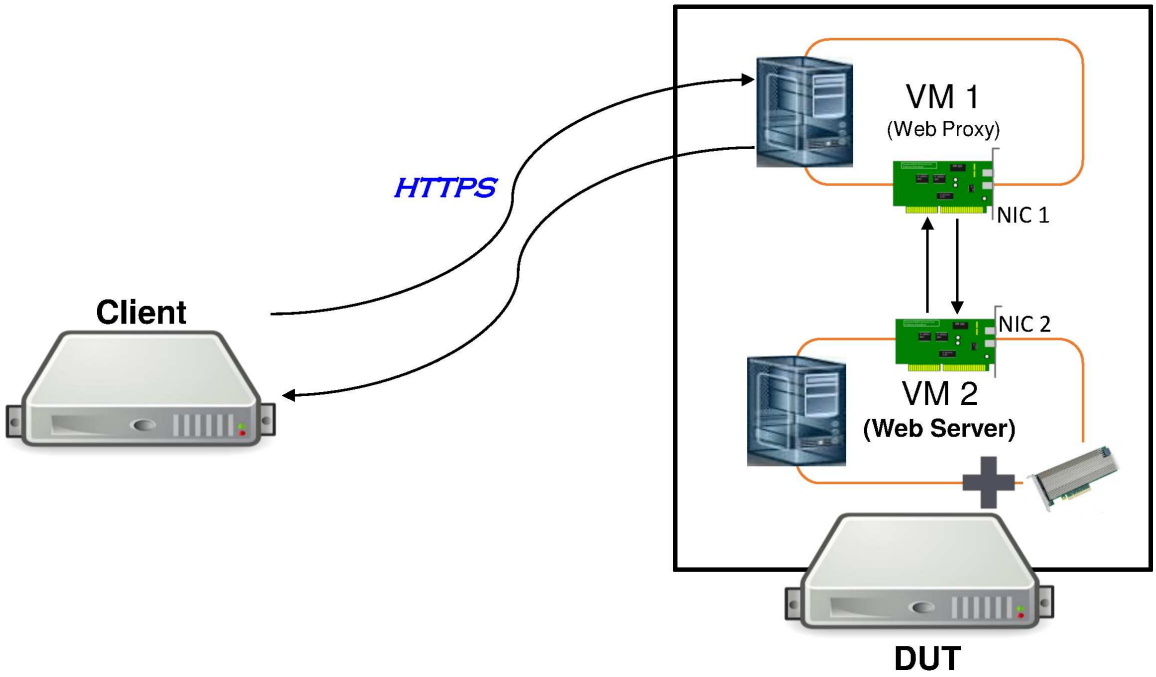

Figure 6.2: Web Server-Proxy-Client Test Setup

The physical server in which the DUT is implemented contains a 10Gbps quad network interfaces, which is connected to the client that has a X520 dual port 10G NIC. The DUT server is also equipped with the Intel QAT adapter with single Intel Xeon CPU (E5-2699v3), and the client is equipped with dual Intel Xeon CPUs (E5-2658v3). In this setup, the maximum theoretical network bandwidth is 10G.

During the experiments, several messages are exchanged between the DUT and client to establish secure connection including data transmission and session termination. Each test-case runs for 200 seconds at a time, while NGINX (version 1.6.2) [17] is running on the DUT to provide a web service using the QAT functions. Performance of DUT during each test run is monitored and recorded by *mpstat* and *top*. *HEF* executes every test case 10 times and collects the results in a remote machine. OpenSSL v1.0.2g is used on each tester machine, and QAT compatible OpenSSL v1.0.1-async is installed on the DUT. OpenSSL libcrypto patch (version 0.4.9-009) is also applied on DUT for the optimal performance.

6.2 Performance Benchmark

After the benchmark, the results are analyzed by Analysis Agent of *HEF*. The agent retrieves the raw data from the results database whenever necessary and conducts comparative analysis. The comparative analysis shows performance gaps between the test environment with- and without-QAT.

Figure 6.3, 6.4, 6.5, and 6.6 show the experimental results in QAT-installed and non-QAT environments with single CPU core. Figure 6.3 and 6.4 are the performance comparison between the two environments for different cipher suites, while Figure 6.5 and 6.6 are the CPU utilization comparison between the two environments while measuring throughput and CPS respectively.

The major contribution made by generating the comparative analysis is that it

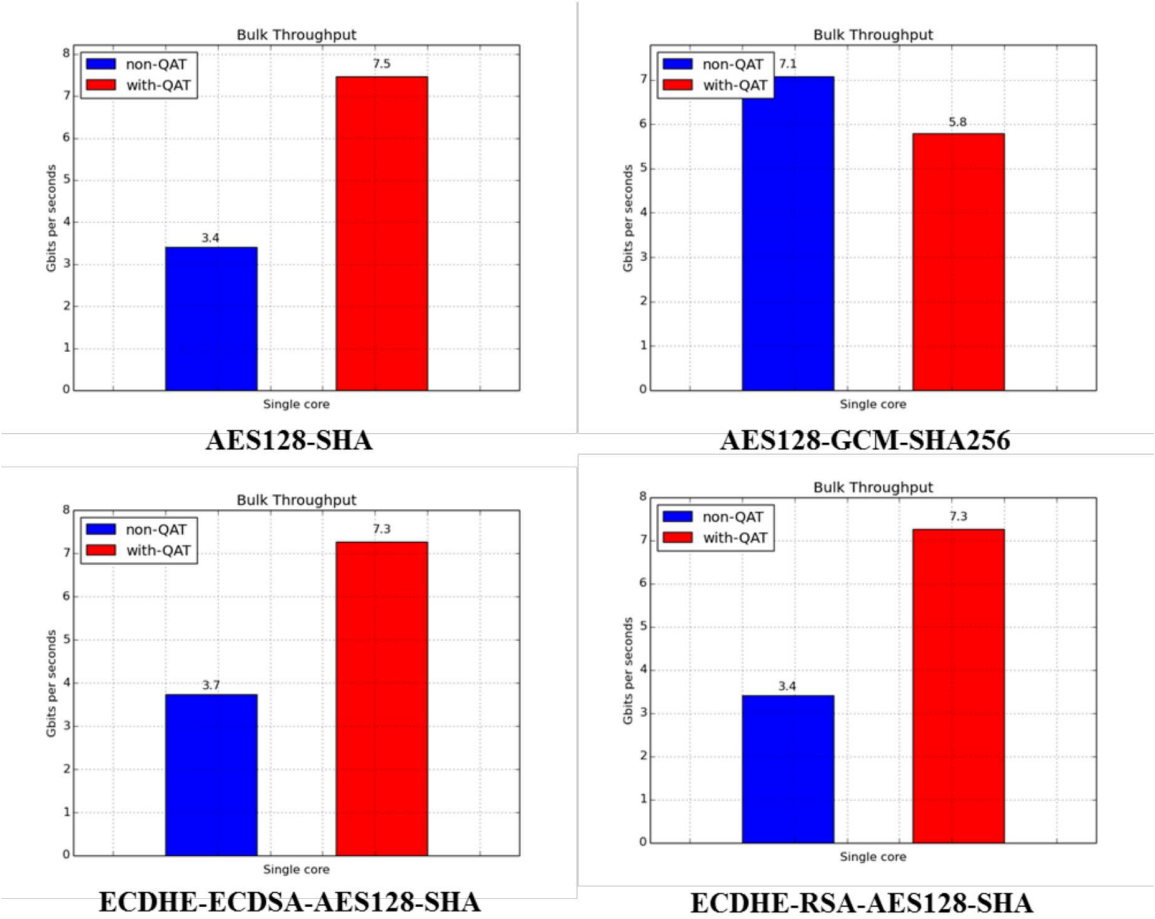

Figure 6.3: Throughput Result with 1 Core in QAT and Non-QAT Environments

provides not only performance measurement but also helps the user identify any bottleneck if exists. For example, the throughput result of AES128-GCM-SHA256 cipher suite in Figure 6.3 indicates that the performance of non-QAT environment exceeds that of QAT-installed one. The result indicates that there might be a possible bottleneck for the specific cipher function of QAT because QAT is the only device that is running AES128-GCM-SHA256 cipher suite. The result means that QAT is not optimized and creates performance overhead for the cipher suite. Therefore, manufacturers (in this case, QAT manufacturers) can identify the bottleneck for their devices from the in-depth analysis generated by *HEF*. The issue is reported to Intel

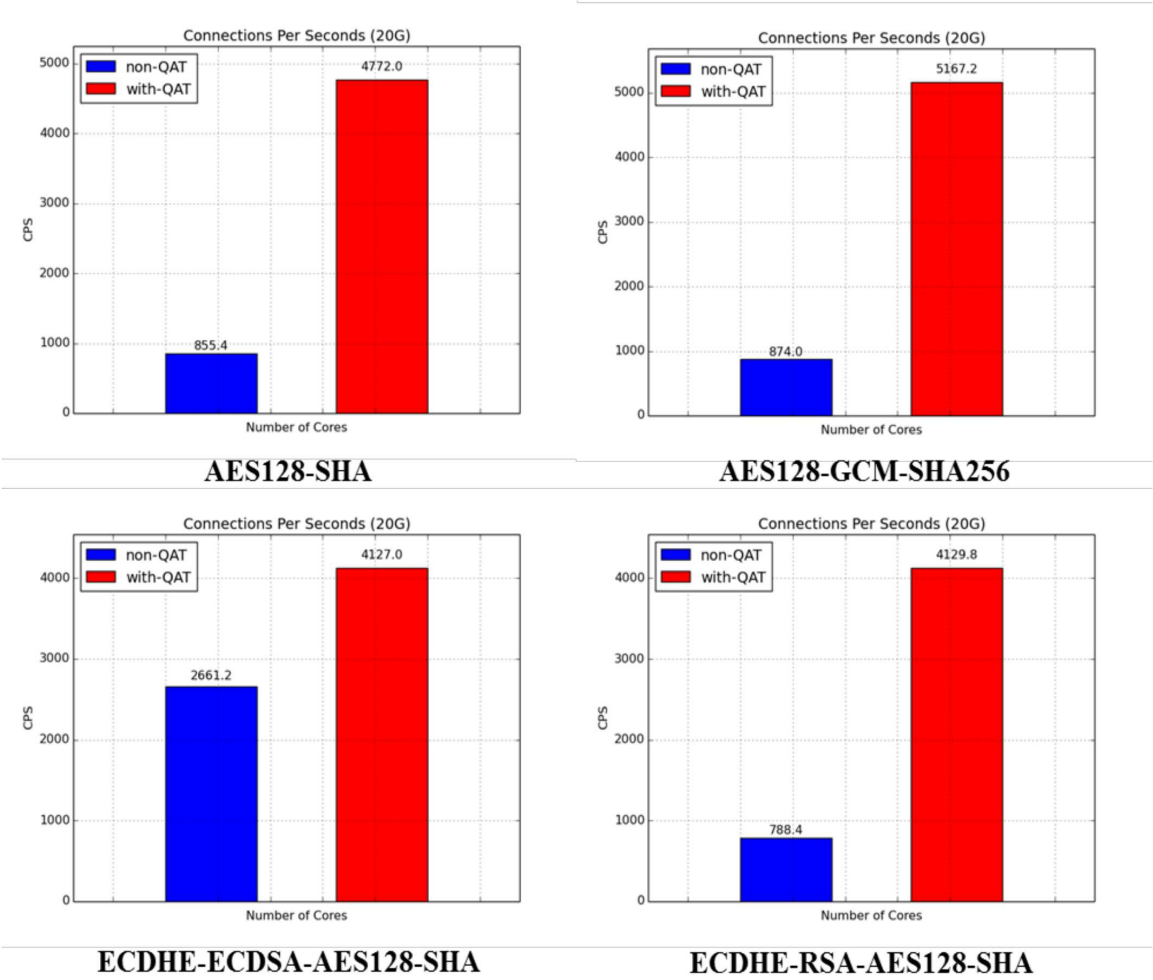

Figure 6.4: CPS Result with 1 Core in QAT and Non-QAT Environments

along with the results generated by *HEF*. The entire experimental results are included in Section 7.

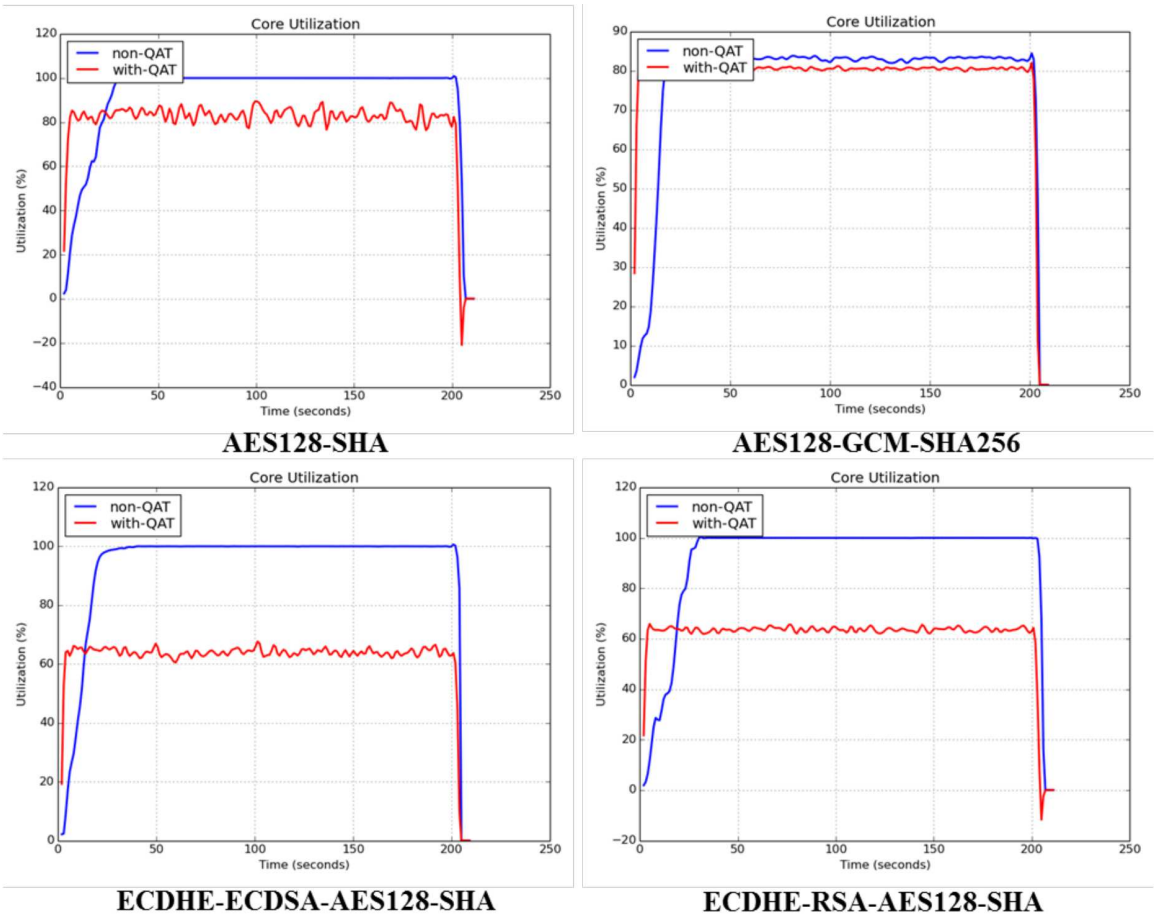

Figure 6.5: CPU Utilization Result for Throughput Experiment with 18 Cores

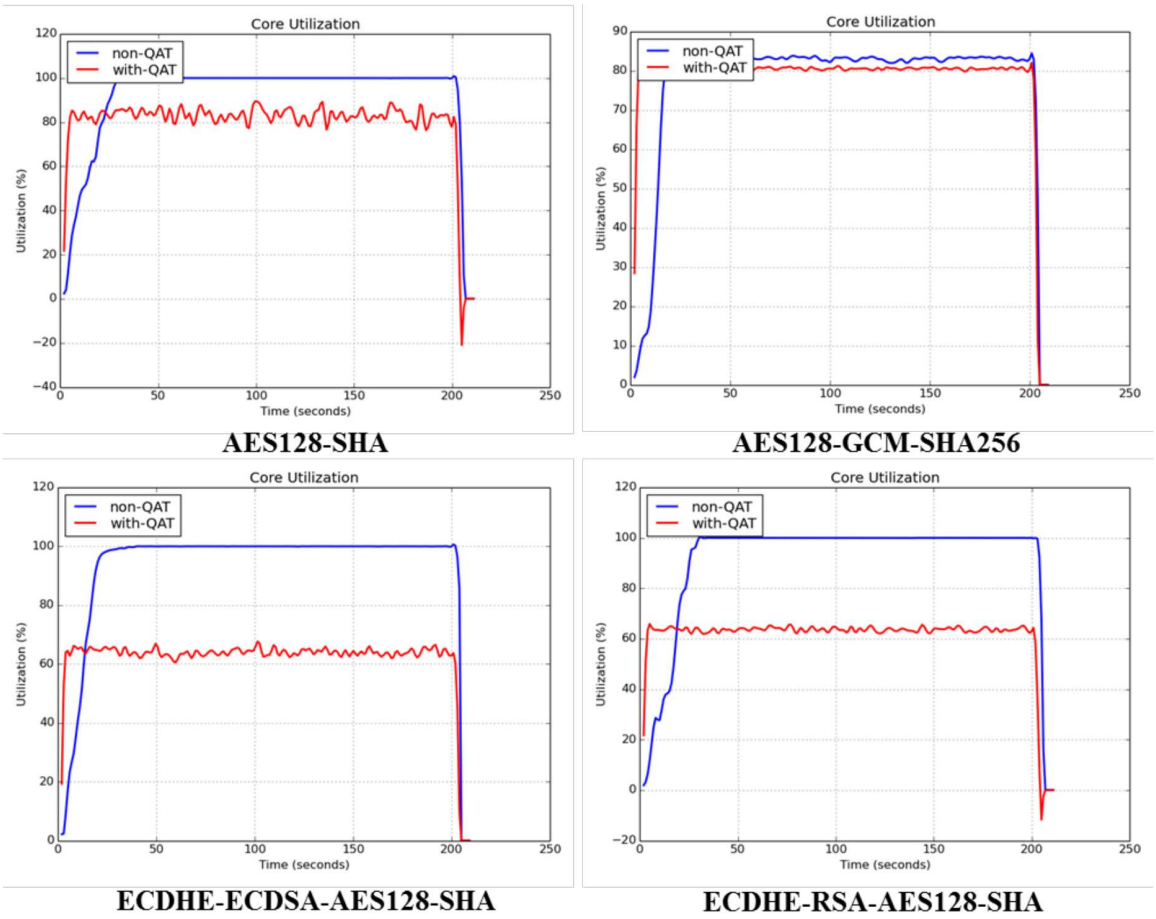

Figure 6.6: CPU Utilization Result for CPS Experiment with 18 Cores

Chapter 7

EXPERIMENTAL RESULTS

7.1 Results in Web Server-Cient Environment

This section provides the result graphs generated by Ananlysis Module of *HEF*. The results presented here are the performance analysis of the web server-client test environment with- and without-QAT. We first conducted the experiment to measure bulk throughput of the web server-client environment. Figure 7.1 and 7.2 illustrates the bulk throughput results obtained from the test environment. Each of four different cipher suites (AES128-SHA, AES128-GCM-SHA256, ECDHE-ECDSA-AES128-SHA, and ECDHE-RSA-AES128-SHA) is used for encryption/decryption while measuring the throughput.

The upper left graph in Figure 7.1 indicates that the throughput result using AES128-SHA has increased from 3.4 to 7.5 Gbps with the help of QAT, thereby achieving significant performance gain (221 % increase). Performance of QAT with ECDHE-ECDSA-AES128-SHA and ECDHE-RSA-AES128-SHA cipher suites also showed significant increase — 197 % and 215 % increase, respectively. However, we observed performance degradation while using AES128-GCM-SHA256 cipher suite — from 7.1 Gbps without QAT to 5.8 Gbps with QAT.

Meanwhile, the throughput with QAT gradually exceeds the throughput without QAT as the number of cores bound to NGINX web server increases (Figure 7.2). The results in general show that the throughput without QAT increases linearly until it reaches its maximum, while the same environment with QAT reaches its maximum with less number of cores. On one hand, the throughput without QAT us-

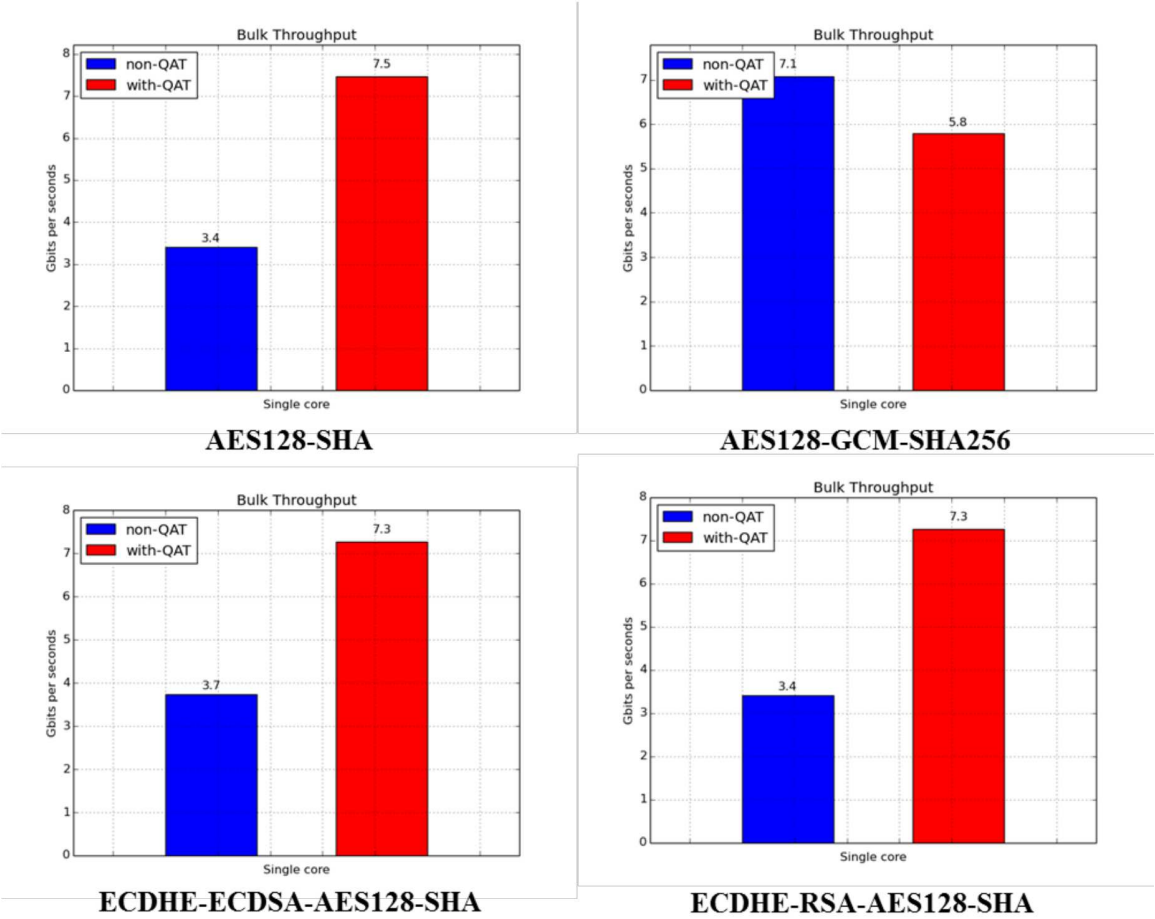

Figure 7.1: Bulk Throughput Results with 1 Core

ing AES128-SHA, ECDHE-ECDSA-AES128-SHA, and ECDHE-RSA-AES128-SHA keeps increasing up to 45.5, 46.3, and 44.6 Gbps utilizing 18 cores. On the other hand, the setup with QAT requires only 8 to 10 cores for its maximum throughput (Figure 7.2). However, the results with AES128-GCM-SHA256 cipher suite shows different patterns. Up to 8 CPU cores, the environment without QAT generated more throughput (20 % more in average)compared to the one with QAT. And then, both setups (with- and without-QAT) reached their maximum at 10 cores. From the observation, we see that QAT can generate better performance for throughput with AES128-SHA, ECDHE-ECDSA-AES128-SHA, and ECDHE-RSA-AES128-SHA

cipher suites. We can also conclude that there is little or no difference in throughput values when AES128-GCM-SHA256 cipher suite is used.

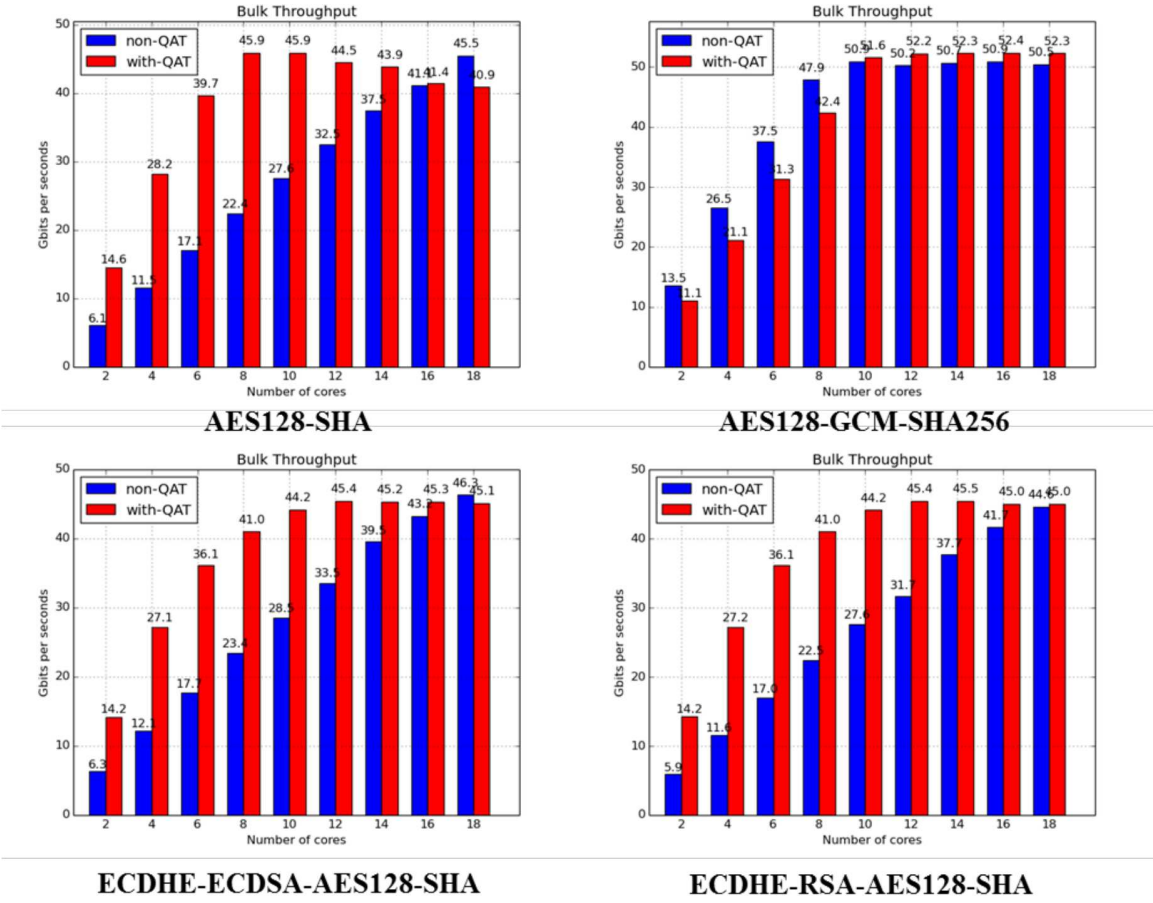

Figure 7.2: Bulk Throughput Results with Multiple Cores

The performance improvement by using QAT is more clear when we look at the CPS results. At most 591 % increase in CPS is observed with AES128-GCM-SHA256 cipher suite (upper right graph in Figure 7.3). The lease increase in CPS by using QAT is 155 % with ECDHE-ECDSA-AES128-SHA cipher suite (lower left graph in Figure 7.3).

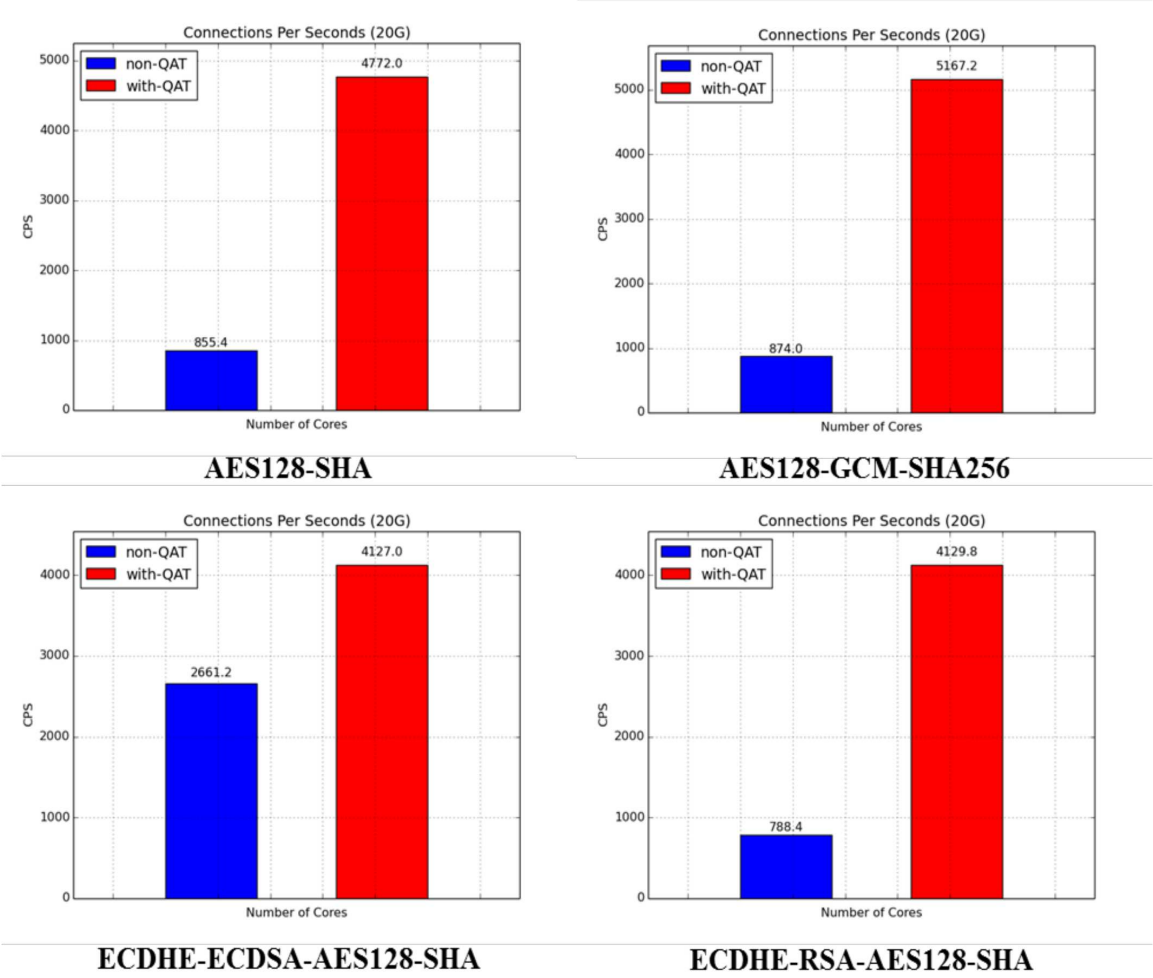

Figure 7.3: CPS Results with 1 Core

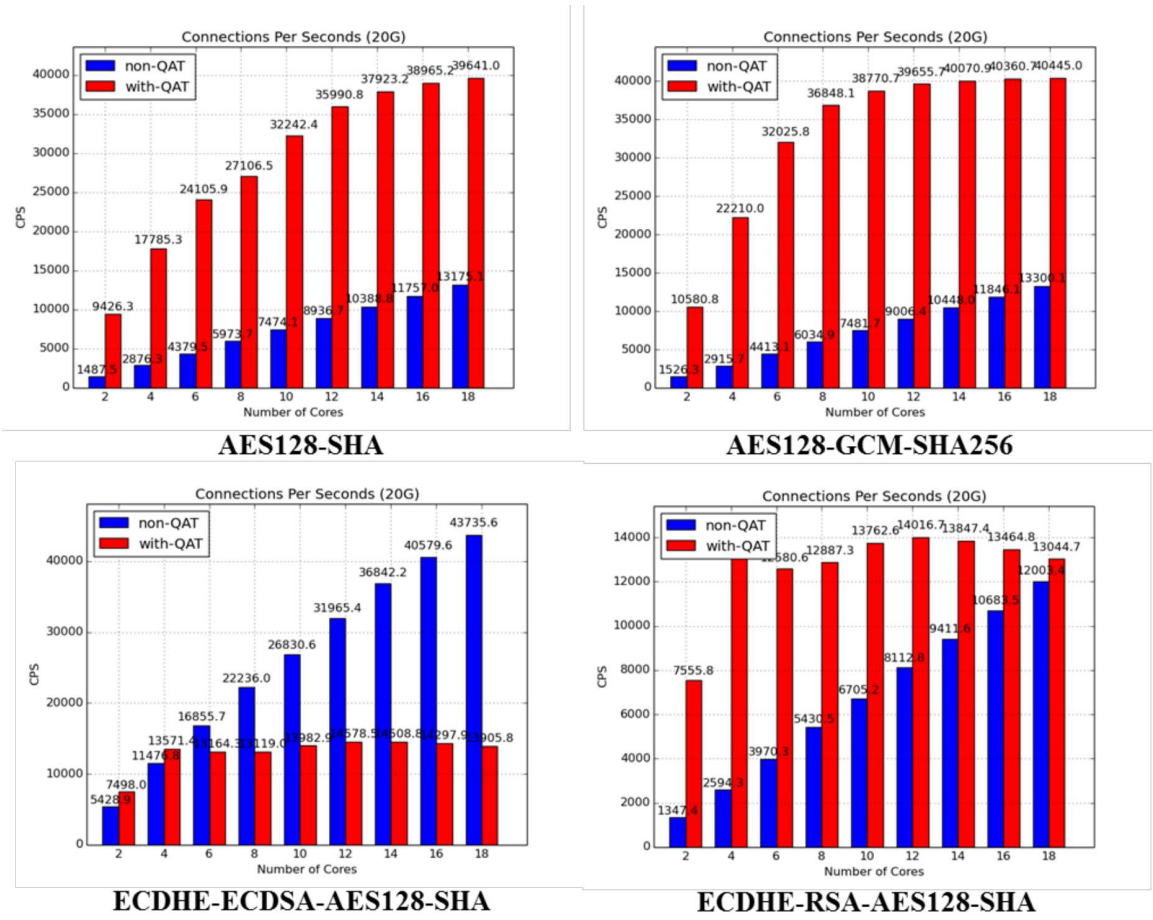

Figure 7.4: CPS Results with Multiple Cores

The results for CPS in multi-core environments are consistent with that of single-core environment, with one exception. With AES128-SHA and AES128-GCM-SHA256 cipher suites, CPS in the setups with QAT reached up to approximately 40,000, while the one without QAT only reached up to 13,000. However, the results for ECDHE-ECDSA-AES128-SHA and ECDHE-RSA-AES128-SHA cipher suites show only 13,000 CPS in maximum even with QAT (Figure 7.4). Especially, the result for ECDHE-ECDSA-AES128-SHA cipher suite shows that the setup *without* QAT outperforms the one *with* QAT (lower left graph in Figure 7.4). We speculate the reason for this result is due to the limited capacity of the single-core QAT adapter

we used in this case study. Better performance can be expected with multi-core QAT adapter because it may be capable of scaling its own performance. This issue is also reported to Intel, along with the one described in Section 6.2.

We also measure CPU utilization in percentage (%). Figure 7.5 and 7.6 illustrates the utilization results measured during the throughput and CPS experiments with 18 CPU cores. Both results show QAT achieved the same or better performance with less CPU resources, while the setup without QAT consumes almost all CPU resources available. The setup with QAT shows maximum reduction in resource consumption by 35 % during the each period of 200 seconds for throughput experiment (Figure 7.5).

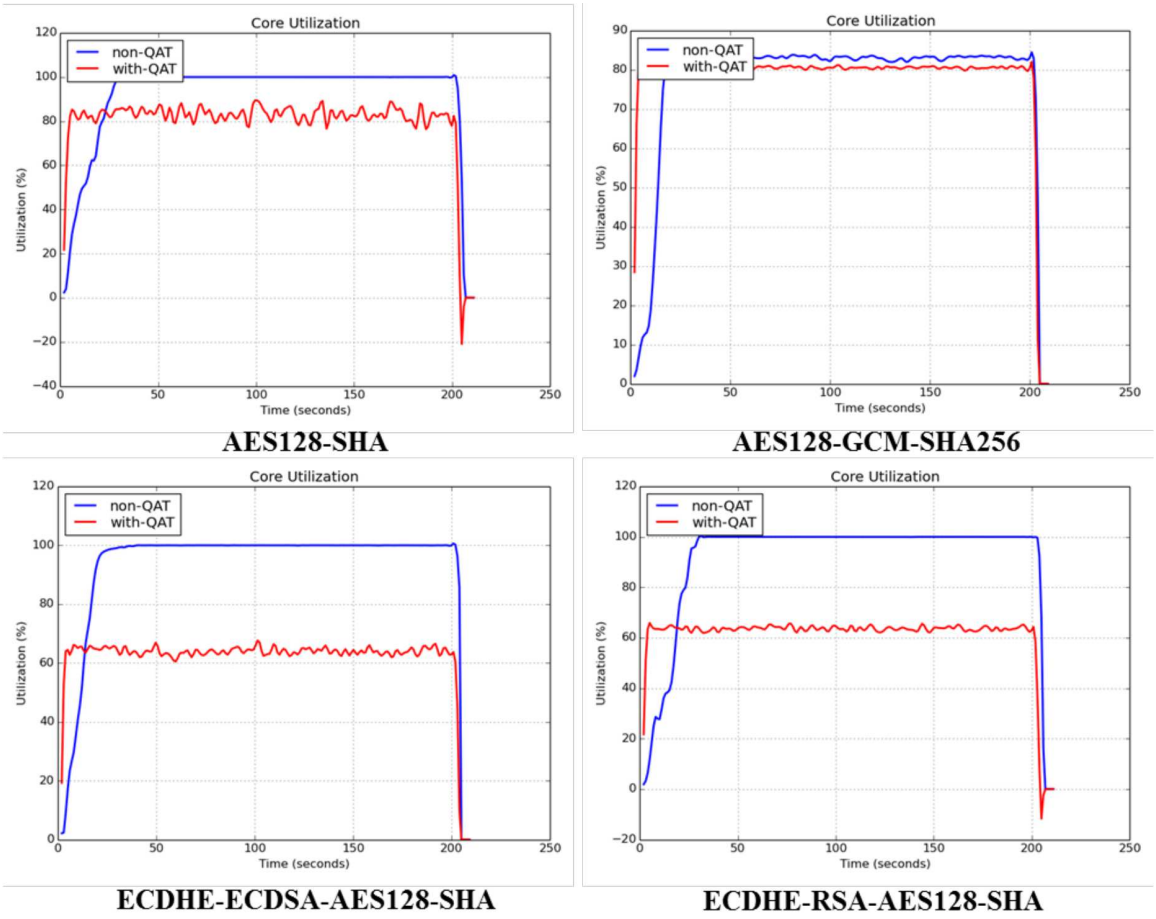

Figure 7.5: CPU Utilization for Throughput with 18 Cores

For CPS experiment, QAT shows even more reduction in CPU resource utilization. While the server without QAT utilized 100 % of the given CPU resources regardless of the cipher suites, the one with QAT utilized at most 80 % of CPU. Figure 7.6 shows that, with QAT, CPU utilization drops to less than 50 %. From the results, we can conclude that the server can conserve CPU resources with the help of QAT, alleviating the overhead of compute-intensive tasks.

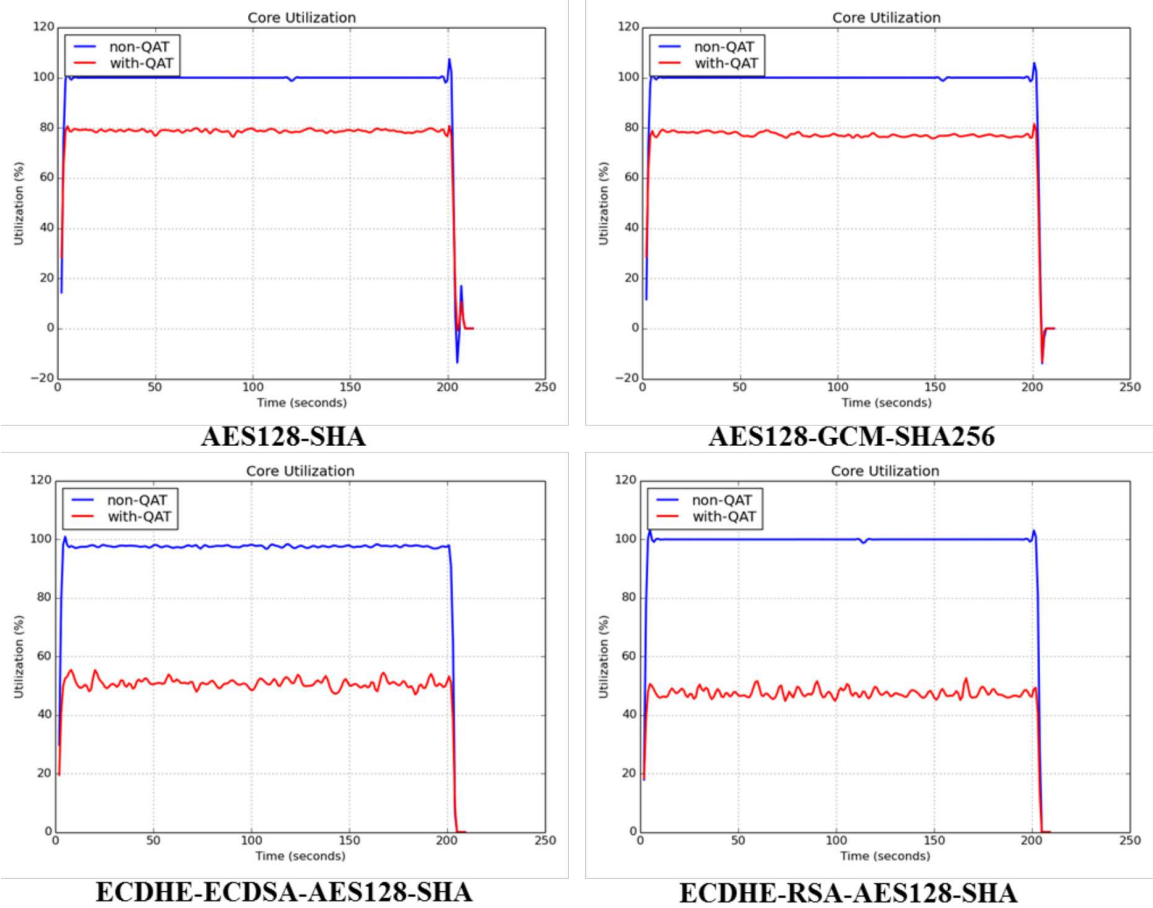

Figure 7.6: CPU Utilization for CPS with 18 Cores

7.2 Results in Web Server-Proxy-Cient Environment

This section also provides the result generated by Analysis Module of *HEF*. The results presented here are the performance analysis of the web server-proxy-client test environment with- and without-QAT. The results from throughput experiment is illustrated in Figure 7.7. The results are consistent with the ones obtained from the web server-client environment; more throughput is generated using QAT when AES128-SHA, ECDHE-ECDSA-AES128-SHA, and ECDHE-RSA-AES128-SHA cipher suites are used. There is at most 158 % increase in throughput when ECDHE-RSA-AES128-SHA cipher suite is used. For ECDHE-ECDSA-AES128-SHA and AES128-SHA, there is 141 %, and 150 % increase in throughput respectively. However, the server without QAT generated more throughput when AES128-GCM-SHA256 cipher suite is used for the same reason described in Section 6.2.

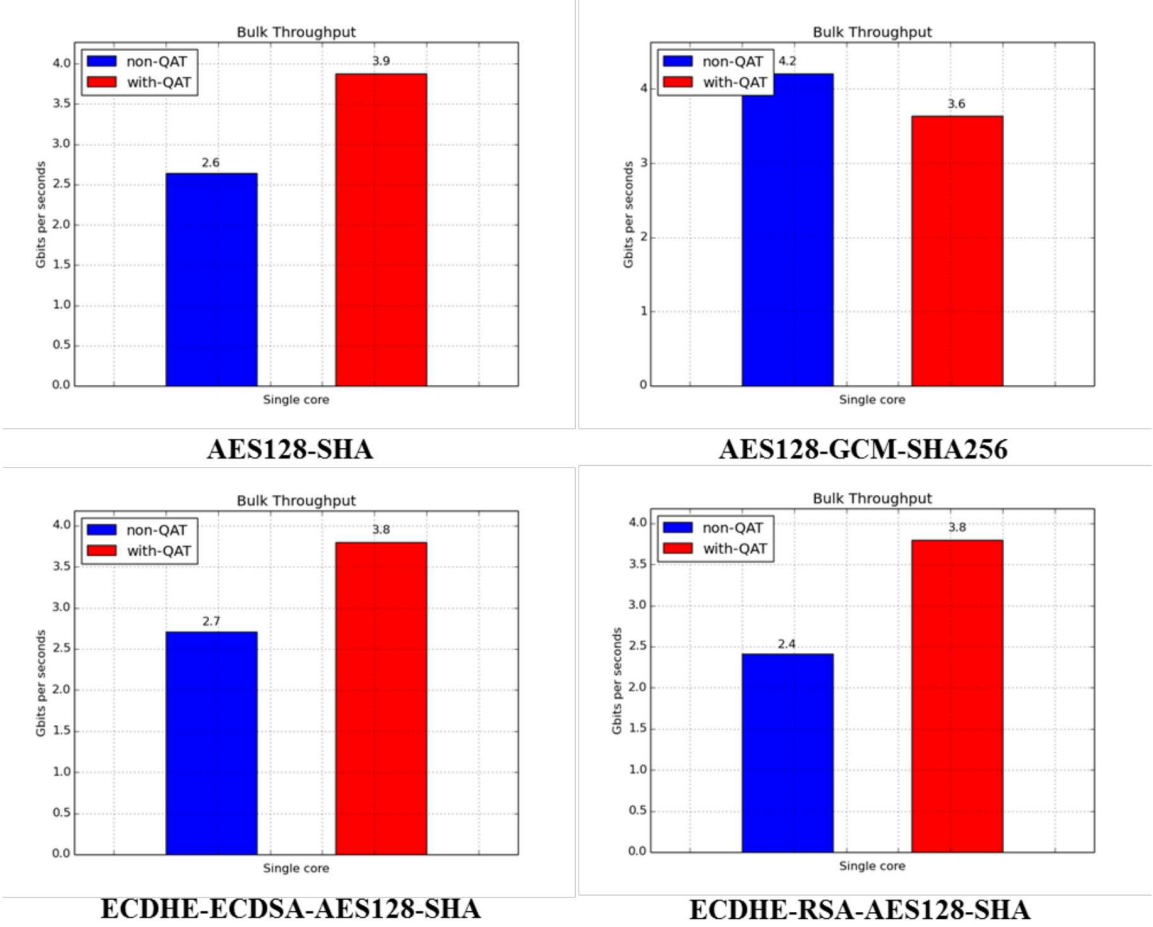

Figure 7.7: Bulk Throughput Results with 1 Core

For multi-core environments, the results for all four cipher suites have identical pattern. Figure 7.8 shows that throughput keeps increasing linearly in the server without QAT as the number of CPU cores increases. Throughput in the environment with QAT reaches its upper limit at lower number of cores (10 $\sim$ 12 cores). In the web server-proxy-client setup, upper limits are in the range of 30 Gbps $\sim$ 38 Gbps, depending on the cipher suites used.

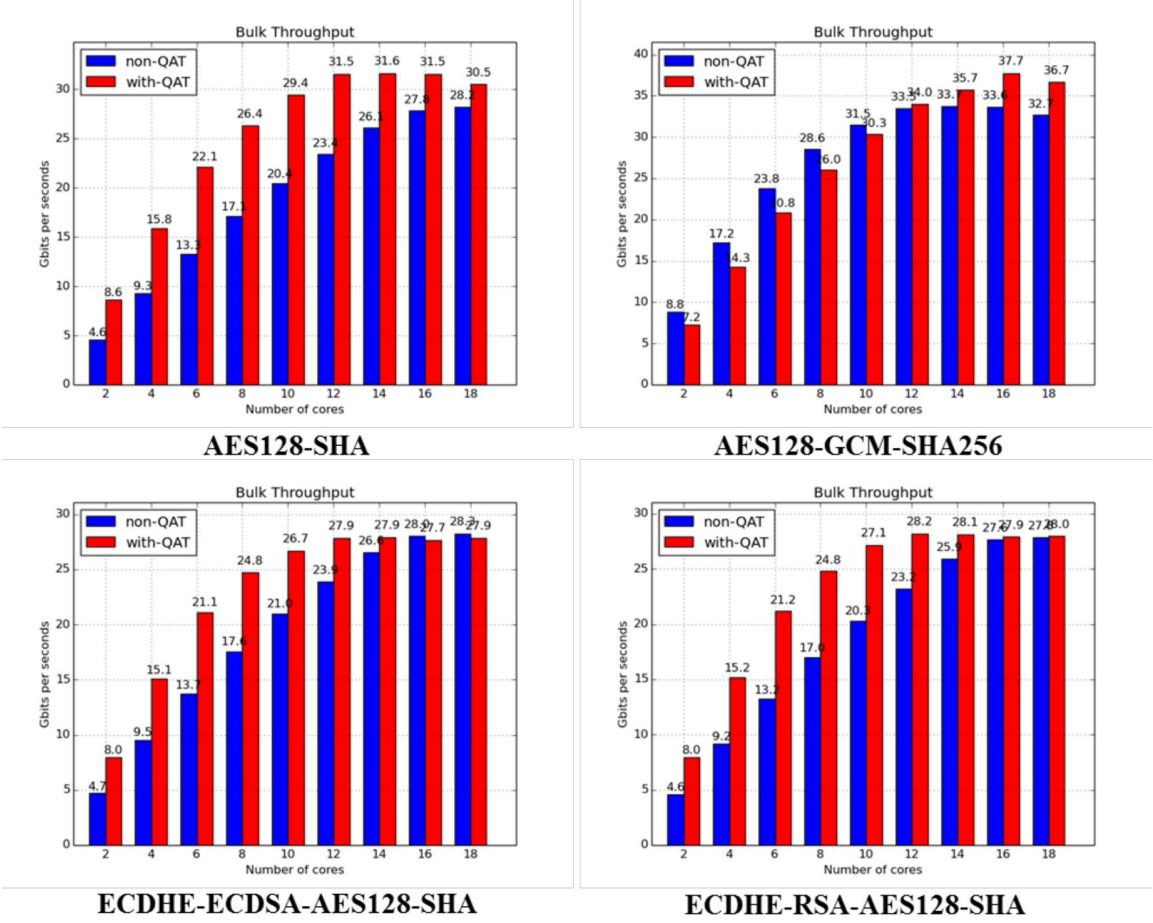

Figure 7.8: Bulk Throughput Results with Multiple Cores

Figure 7.9 shows that significant improvements in CPS values are made by using QAT. For AES128-GCM-SHA256 cipher suite, there is 567 % increase in CPS from 864.2 to 4903.3 CPS. The least increase in CPS are observed in ECDHE-ECDSA-AES128-SHA cipher suite (157 % increase). The results indicate that QAT can handle thousands of more concurrent connections in a second, even with a single CPU core. The results also mean using QAT can achieve significant performance increase in terms of CPS.

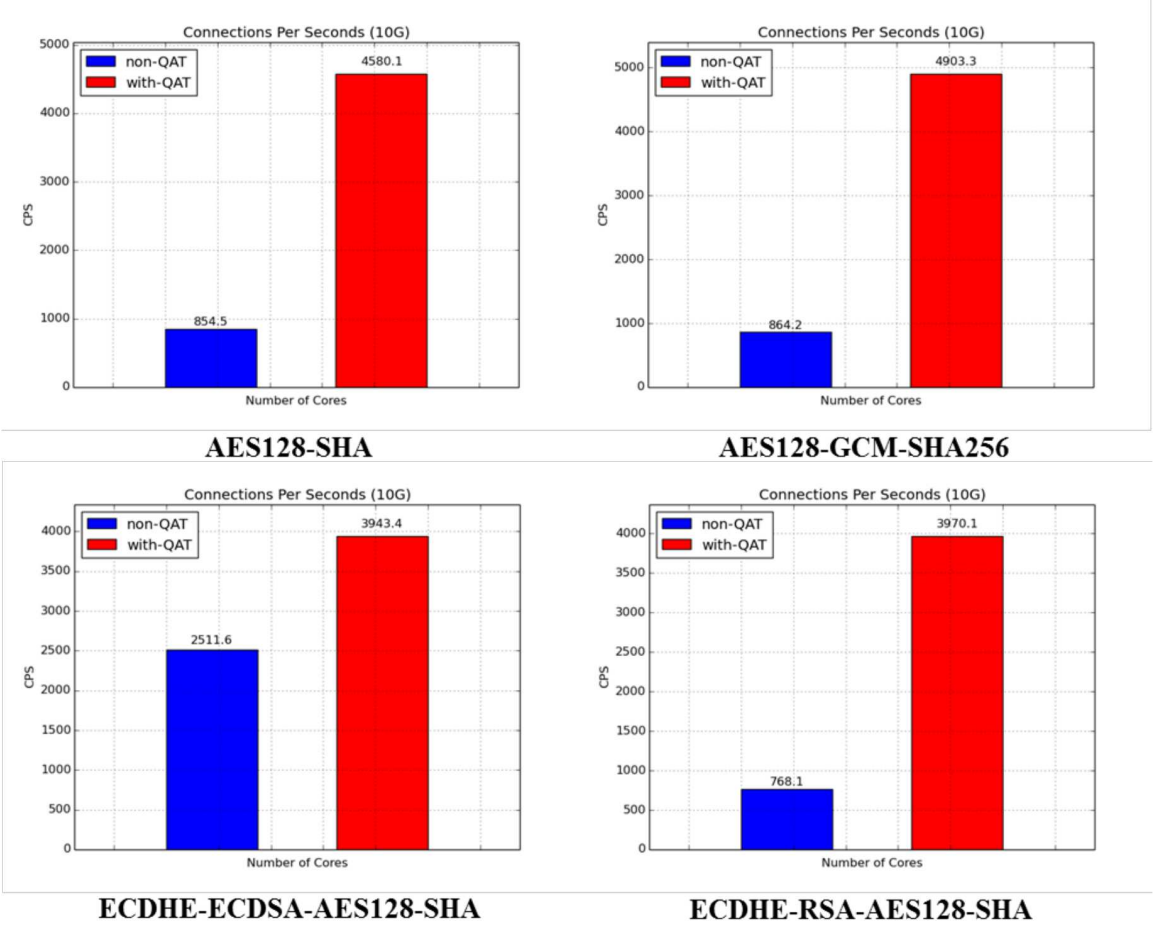

Figure 7.9: CPS Results with 1 Core

Figure 7.10 are the results generated by *HEF* for CPS in multi-core environments. In general, CPS under the environment without QAT increased linearly up to 18 cores, while CPS in the environment with QAT reaches the upper limit at lower number of cores ($4 \sim 8$ cores). In ECDHE-ECDSA-AES128-SHA, however, the maximum CPS generated QAT (approximately 11,000 CPS) is less than that of without-QAT setup (approximately 26,000 CPS). This result also stems from the limitation of single-core QAT adapter employed in this experiment, as explained in Section 7.1. Note that, in the setup without QAT, CPS reaches its maximum value at 12 cores. What is interesting is that, after reaching its maximum at 12 cores, CPS decreases up to 18

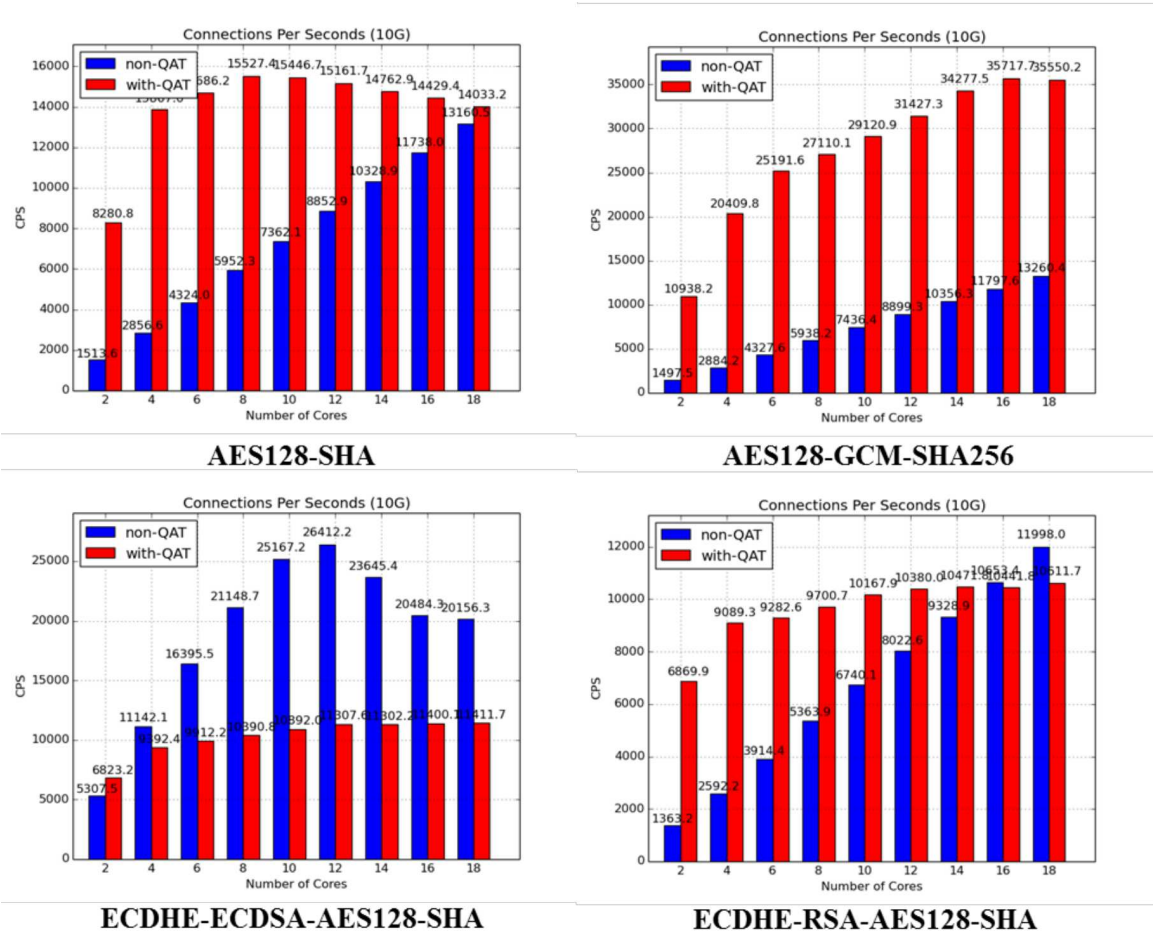

Figure 7.10: CPS Results with Multiple Cores

cores. The fluctuation in CPS indicates that there may exist some problems in proxy, such as load-balancing with multiple CPU cores.

The results in Figure 7.11 are also consistent with the ones from web server-client test environment. While the DUT without QAT utilizes almost 100 % of the CPU resources regardless of the cipher suites, DUT with QAT utilizes only 74 $\sim$ 85 % of the resource, reducing the CPU utilization by maximum 26 %. Figure 7.12 shows CPU utilization results for CPS experiments with 18 cores. The results indicate that the DUT with QAT can outperform the without-QAT setup using even less CPU resources, when dealing with CPS. The maximum reduction in CPU utilization is

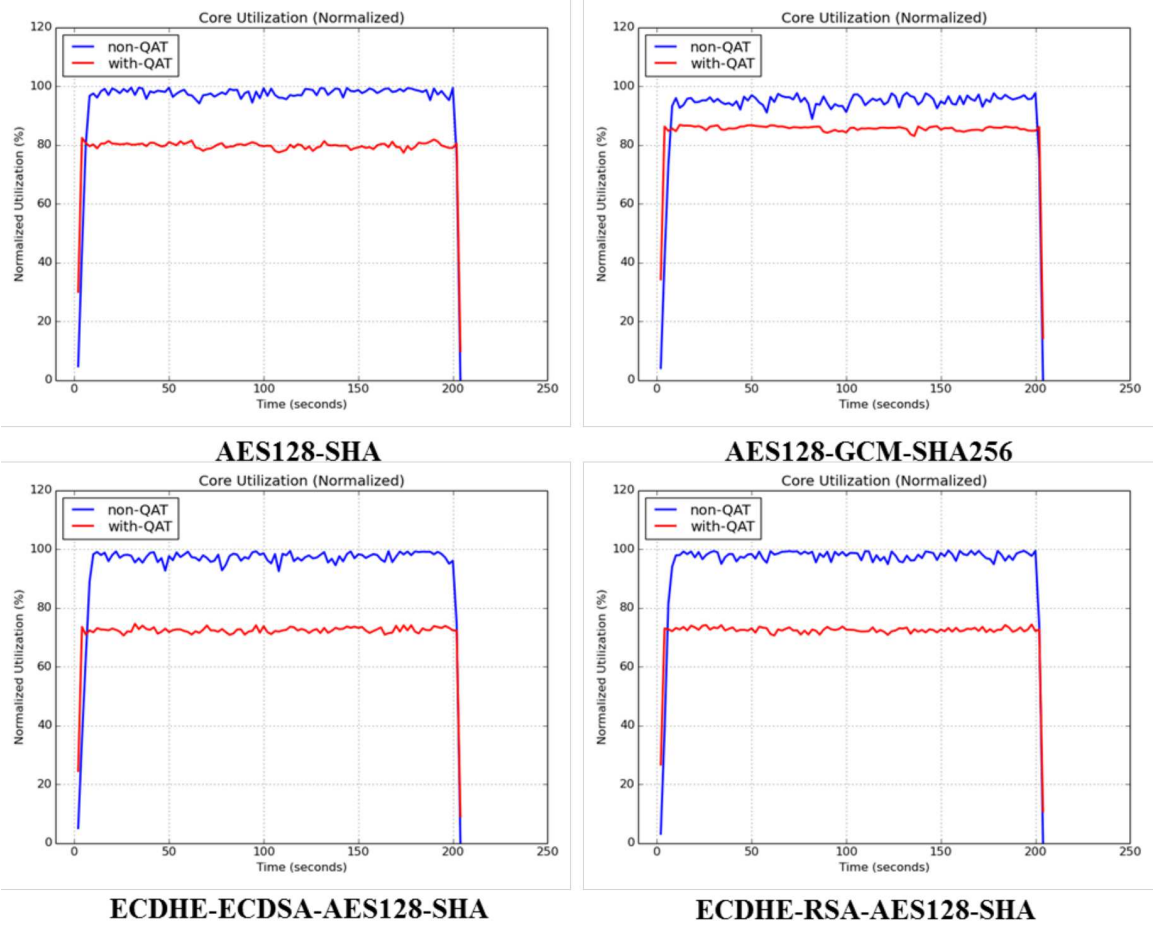

Figure 7.11: CPU Utilization for Throughput with 18 Cores

57 %, when ECDHE-RSA-AES128-SHA cipher suite is used. Another interesting result generated is the lower left graph in Figure 7.12 (for ECDHE-ECDSA-AES128-SHA cipher suite); DUT without QAT showed considerably huge fluctuation in CPU utilization, with 57.7 % utilization in average.

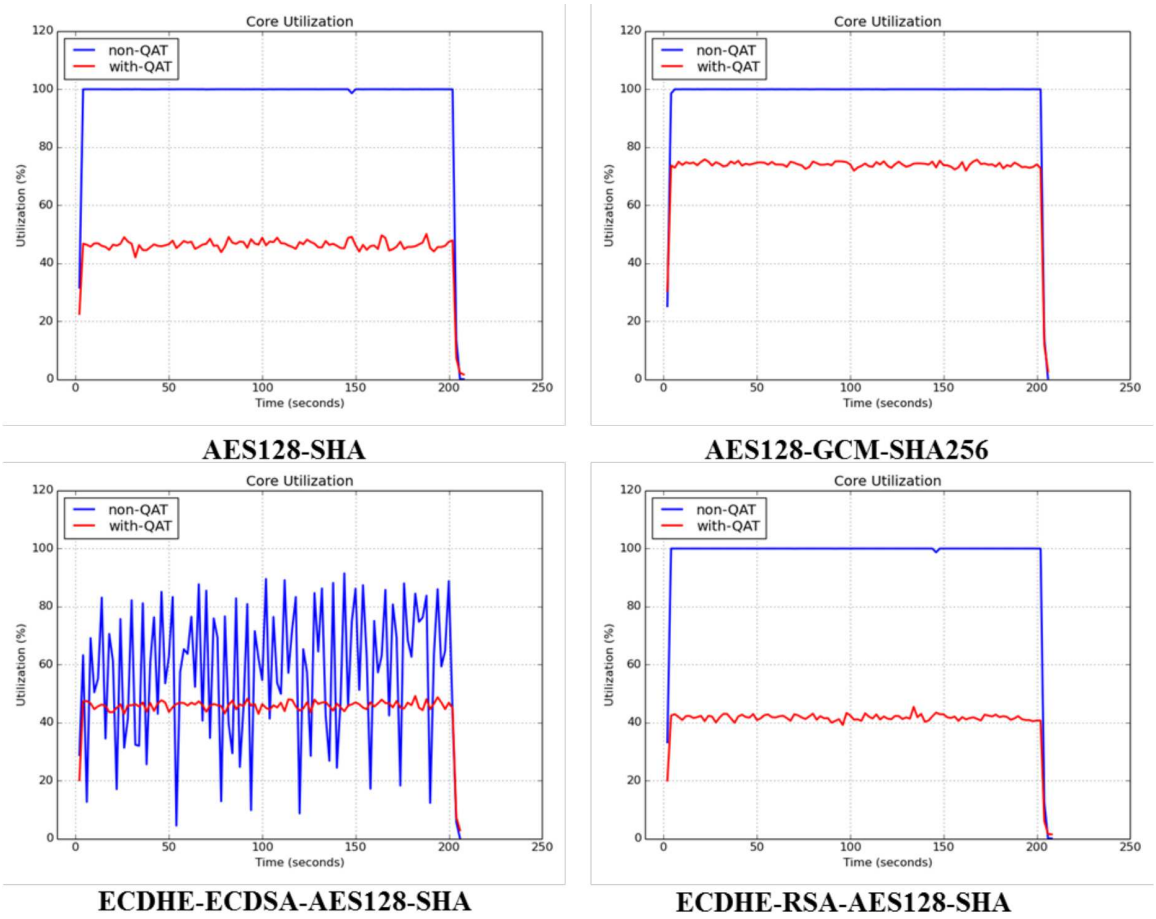

Figure 7.12: CPU Utilization for CPS with 18 Cores

Chapter 8

RELATED WORKS

In this Chapter, other studies related to *HEF* are introduced. First, the configuration of test environments for the study case of QAT is influenced by Intel Open Network Platform [10]. ONP is a server reference architecture that is designed to support network function virtualization (NFV) [8] and software-defined network [12].

VANFC devised by Smolyar et al. [21] proposes a new SR-IOV network interface cards (NIC) design that can prevent a certain attack using Ethernet pause frames. The authors first demonstrated that they can successfully exploited the vulnerability in NIC with SR-IOV caused by the lack of flow control. To address this issue, the authors implemented a prototype of virtual functions that makes the NIC aware of the type of flow. The virtual functions also include the filtering function that can filter certain flow at zero performance overhead.

NFV-VITAL [5] is a virtualized network function (VNF) characterization framework that characterizes VNFs based on user preference and available resources. A simple performance benchmark is also performed to show the level of performance overhead created by using NFV-VITAL. The idea of using the user-defined input in *HEF* is inspired by the design of NFV-VITAL. However, NFV-VITAL is only applicable to certain VNF platform (Clearwater IMS) and is not directly related to the problems address in this thesis.

Chapter 9

DISSCUSSION

In the previous chapters, we discuss our motivation, approach, implementation, and case study to demonstrate the effectiveness of *HEF*. In this chapter, limitations and assumptions of *HEF* are discussed.

HEF possesses certain limitations. Even though it is designed to be generic and automated framework for any HAS hardware, the user has to define test cases and metrics for each device if required. Configuration of test environments may also require certain degree of user intervention. For example, one of the software requirements for QAT benchmark is QAT compatible OpenSSL, which must be patched by modifying the source code. The task cannot be automated because it is the task required for QAT specifically. Another task that cannot be automated is enabling intel VT-d from system booting menu to use SR-IOV [11], and thus, it must be performed by the user. Therefore, it requires the user to prepare certain inputs and may also require manual preparation of test environments depending on the HAS deviced evaluated.

In addition, *HEF* is created with certain assumptions. HAS technology concerned in this thesis is limited to the devices that are installed in standard, high-volume servers. The environments in which those servers are used include data centers, cloud service providers, and high performance computing setups. Therefore, different HAS technologies designed for other platforms, such as TPM used in embedded systems, are not in the scope of this thesis.

Chapter 10

CONCLUSION

In this thesis, we proposed a design of a generic testbed framework for HAS devices. We name this HAS performance evaluation framework as *HEF*. *HEF* is devised in an attempts to address the lack of universal test environment that can evaluate the performance of a HAS device. By designing and implementing *HEF*, we demonstrated functionalities of it as a generic framework. Also, efficacy of *HEF* as a tool for in-depth comparative performance analysis is proved.

Finally, the contributions of this thesis are summarized as follows:

1. A generic performance analysis framework for HAS devices is designed in this thesis. This HAS performance evaluation framework (*HEF*) minimizes the user intervention by automating the performance benchmark. Thus, *HEF* reduces the time and effort required to prepare a testbed for any HAS device.

2. *HEF* is designed to be universal. In other words, the framework is not device-specific and is applicable to any HAS device for standard high-volume servers. To this end, *HEF* is designed to take user-defined inputs for different test metrics, test-cases, and software tools required.

3. An instance of *HEF* is implemented and evaluated for its efficacy, using Intel QAT device. *HEF* generates comparative performance analysis by running benchmark in the same testbed with- and without-QAT. The results generated by *HEF* provides an in-depth analysis that helps network vendors/administrators identify hidden bottlenecks in the target device.

4. By conducing the case study using *HEF*, we identified unknown bottlenecks in certain cipher functions provided by QAT device. The issues are reported to Intel and it acknowledged the issue submitted by us.

REFERENCES

[1] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. viommu: efficient iommu emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011.

[2] Django Armstrong and Karim Djemame. Performance issues in clouds: An evaluation of virtual image propagation and i/o paravirtualization. *The Computer Journal*, page bxr011, 2011.

[3] Ahmed M Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 38–49. ACM, 2010.

[4] Scott Bradner and Jim McQuaid. Rfc 2544. *Benchmarking methodology for network interconnect devices*, 1999.

[5] Lianjie Cao, Puneet Sharma, Sonia Fahmy, and Vinay Saxena. Nfv-vital: A framework for characterizing the performance of virtual network functions. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 93–99. IEEE, 2015.

[6] CISCO. The zettabyte eratrends and analysis. `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.pdf`, 2016.

[7] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[8] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[9] Intel. How intel quickassist technology accelerates network function use cases. `https://software.intel.com/en-us/articles/how-intel-quickassist-technology-accelerates-nfv-use-cases`. [Online; accessed 2017-1-21].

[10] Intel. Intel open netowrk platform (intel onp). `https://www-ssl.intel.com/content/www/us/en/communications/intel-open-network-platform.html`. [Online; accessed 2017-1-13].

[11] Intel. Intel sr-iov configuration guide. `https://www-ssl.intel.com/content/dam/www/public/us/en/documents/technology-briefs/xl710-sr-iov-config-guide-gbe-linux-brief.pdf`.

[12] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, 2013.

[13] S. Luo, Z. Lin, X. Chen, Z. Yang, and J. Chen. Virtualization security for cloud computing service. In *2011 International Conference on Cloud and Service Computing*, pages 174–179, Dec 2011.

[14] Peter Mell and Tim Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[15] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Kang G Shin, et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 137, 2007.

[16] R. Perez, L. van Doorn, and R. Sailer. Virtualization and hardware-based security. *IEEE Security Privacy*, 6(5):24–31, Sept 2008.

[17] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[18] Farzad Sabahi. Virtualization-level security in cloud computing. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 250–254. IEEE, 2011.

[19] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226. IEEE, 2010.

[20] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Van Doorn, John Linwood Griffin, Stefan Berger, Reiner Sailer, Enriquillo Valdez, Trent Jaeger, et al. shype: Secure hypervisor approach to trusted virtualized systems. *Techn. Rep. RC23511*, 5, 2005.

[21] Igor Smolyar, Muli Ben-Yehuda, and Dan Tsafrir. Securing self-virtualizing ethernet devices. In *USENIX Security*, pages 335–350, 2015.

[22] Kuyoro So. Cloud computing security issues and challenges. *International Journal of Computer Networks*, 3(5):247–55, 2011.

[23] Jakub Szefer, Eric Keller, Ruby B Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 401–412. ACM, 2011.

[24] Paul C Van Oorschot, Anil Somayaji, and Glenn Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, 2(2):82–92, 2005.

[25] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *International Workshop on Recent Advances in Intrusion Detection*, pages 158–177. Springer, 2010.

[26] Xiaolin Wang, Yan Sang, Yi Liu, and Yingwei Luo. Considerations on security and trust measurement for virtualized environment. *JoC*, 2(2):19–24, 2011.

[27] WEI. Changing the security paradigm with hardware-assisted security. `http://blog.wei.com/changing-the-security-paradigm-with-hardware-assisted-security`, 2016. [Online; accessed 2016-11-24].