HARDWARE-BASED SOLUTIONS FOR SECURING USERS' DATA IN PUBLIC CLOUDS

BY

MOHAMMED AL-ASALI

A Dissertation Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

COMPUTER SCIENCE AND ENGINEERING

MAY 2017

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Mohammed Al-Asali** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING.**

Dr. Muhammad E. S. Elrabaa
(Advisor)

عادل

Dr. Adel F. Ahmed
Department Chairman

Marwan

Dr. Marwan H. Abu-Amara
(Member)

Prof. Salam A. Zummo
Dean of Graduate Studies

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

Sadiq Sait · M

Prof. Sadiq M. Sait
(Member)

14/9/17
Date

Dr. Aiman H. El-Maleh
(Member)

Tarek Hel

Dr. Tarek Helmy El-Basuny
(Member)

To

my parents

ACKNOWLEDGMENTS

I would like to thank King Fahd University of Petroleum and Minerals for offering me the opportunity to pursue my graduate and undergraduate studies and for supporting this research.

I would like to express my deep appreciation and heartfelt gratitude to my advisor Dr. Muhammad Elrabaa for his invaluable guidance and infinite patience. He has helped me improve my research and technical writing skills. I have learned a lot of valuable lessons from him and I hope to continue working with him in future.

I would like to thank my committee members Dr. Marwan Abu-Amara, Prof. Sadiq Sait, Dr. Aiman El-Maleh, and Dr. Tarek Helmy EL-Basuny for their valuable comments and suggestions. I especially thank Dr. Marwan Abu-Amara for his valuable time and for his reviews, support and guidance in the field of data security, and Prof. Sadiq Sait for his comments and advices; I learnt a lot while working with him in the CCITR center.

I am thankful to my colleagues and friends: Amran Al-Aghbari, Feras Chikh Oughali, Asim Al-Mekhlafy, Zaid alyafey, Abdullah Al-Sermy, Mohammed Yaqot and Abdullah Alqubalee for their support and valuable information. I also thank Ken Eguro from Microsoft research for his suggestions and comments and Jonathan Dawson for his time in troubleshooting the Chips2.0 compiler and the Ethernet controller.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AES	**:**	Advanced Encryption Standard
AIK	**:**	Attestation Identity Keys
ASIC	**:**	Application-Specific Integrated Circuit
BRAM	**:**	Block RAM
CA	**:**	Certificate Authority
CLB	**:**	Configurable Logic Block
CoT	**:**	Chain of Trust
CP	**:**	Cloud Provider
DCM	**:**	Digital Clock Manager
DH	**:**	Diffie–Hellman–Merkle Key Exchange
DMA	**:**	Direct Memory Access
DSP	**:**	Digital Signal Processing
EK	**:**	Endorsement Key
EPC	**:**	Enclave page cache
EPCM	**:**	Enclave page cache metadata
FHE	**:**	Fully Homomorphic Encryption
FIPS	**:**	The Federal Information Processing Standard
FPGA	**:**	Field Programmable Gate Arrays
FV	**:**	FPGA Vendor
HDL	**:**	Hardware Description Language
HMAC	**:**	Hashed Message Authentication Code

IaaS	**:**	Infrastructure as a Service
IC	**:**	Integrated Circuit
ICAP	**:**	Internal Configuration Access Port
ICMP	**:**	Internet Control Message Protocol
IO	**:**	Input/Output
IoT	**:**	Internet of Things
IP	**:**	Intellectual Property
ITRS	**:**	International Technology Roadmap for Semiconductors
KPA	**:**	Known-Plaintext Attacks
KVM	**:**	Kernel-based Virtual Machine
DRTM	**:**	Dynamic Root of Trust Measurement
LUTs	**:**	Lookup Tables
MAC	**:**	Message Authentication Code
MI	**:**	Malicious Insider
MiM	**:**	Man-in-the-Middle
MODEXP	**:**	Modular Exponentiation Block
NVM	**:**	Non-Volatile Memory
OEM	**:**	Original Equipment Manufacturer
OTP	**:**	One-Time-Programmable
PaaS	**:**	Platform as a Service
PCR	**:**	Platform Configuration Registers
PFS	**:**	Perfect Forward Secrecy
PRE	**:**	Proxy Re-Encryption

PRM	**:**	Processor Reserved Memory
PUFs	**:**	Physically Unclonable Functions
ROT	**:**	Root of Trust
SaaS	**:**	Software as a Service
SGX	**:**	Software Guard Extensions
SHA	**:**	Secure Hash Algorithm
ACM	**:**	Authenticated Code Module
SMM	**:**	System Management Mode
SPRE	**:**	Symmetric Proxy Re-Encryption
SRK	**:**	Storage Root Key
SRTM	**:**	Static Root of Trust for Measurement
SSL	**:**	Secure Socket Layer
TA	**:**	Trusted Authority
TC	**:**	Trusted Computing
TCCP	**:**	Trusted Cloud Computing Platform
TCG	**:**	Trusted Computing Group
TED	**:**	Trusted Extension Device
TPM	**:**	Trusted Platform Module
TXT	**:**	Trusted Execution Technology
User	**:**	a person who uses the on-cloud FPGA. It is interchangeable with client, and client application.
Verilog	**:**	a hardware description language
VHSIC	**:**	VHSIC:Very High Speed Integrated Circuit

ABSTRACT

Full Name : Mohammed Abdulqaher Ahmed Al-Asali

Thesis Title : Hardware-Based Solutions for Securing Users' Data in Public Clouds

Major Field : Computer Science and Engineering

Date of Degree : May 2017

Traditional software-based protection methods are insecure against cloud operators/service providers related attacks. This is due to the fact that cloud service providers physically own the hardware that hosts users' data and computation. Therefore, cloud computing is not achieving maximum growth since sensitive data are not going to be processed in the cloud. A hardware solution is the most valid method that would possibly tackle the problem and expand the use of cloud computing paradigm. In the literature, protecting users' data in the cloud has been an active research area. Significant progress has been made in securing clients' data in the cloud in the last few years. However, existing research either is mostly concerned with traditional attacks that are not targeting the new emerging threat (i.e. securing data from cloud providers and other users in the cloud) or lacks the practicality in the multi-tenant environment or suffering from security weaknesses and large performance overhead. In this dissertation, we propose FPGA-based solutions for securing users' data from cloud providers and other various kinds of attacks. The proposed solutions are suitable for the multi-tenant nature of the cloud and are practical in terms of cost and performance. The proposed solutions in this dissertation can target two primary areas: (1) securing sensitive data that are owned by a client who performs the computation on his data in the cloud, (2) securing sensitive data that are aggregated from multiple sources and processed in the cloud such as internet of things (IoT) data that is collected from IoT devices. We propose a secure way to aggregate and process such data in the cloud and give its software and FPGA implementation details. The results show that the proposed solutions integrate well with other cloud resources and can boot 15 times faster than booting a medium-size conventional virtual machine (VM) on the same cloud and their performance is comparable to a software processing plaintext data. For secure IoT data processing in the cloud, the results also show that our proposed solution is efficient in terms of resources and performance.

ملخص الرسالة

الاسم الكامل: محمد عبدالقاهر احمد العسلي

عنوان الرسالة : حلول هاردوير لحماية بيانات المستخدم في الحوسبة السحابية العامة

التخصص: علوم وهندسة الحاسب الآلي

تاريخ الدرجة العلمية: مايو 2017

طرق الحماية التقليدية القائمة على البرمجيات غير آمنة ضد مشغلي السحابة والهجمات ذات الصلة. ويرجع ذلك إلى حقيقة أن مقدمي الخدمات السحابية يمتلكون الأجهزة التي تستضيف بيانات المستخدمين والتي تقوم باداء العمليات عليها. ولذلك، فإن الحوسبة السحابية لا تحقق أقصى قدر من النمو حيث ان البيانات الحساسة لن يتم وضعها في السحابة الالكترونية. استخدام الهاردوير هو الأسلوب الأكثر فعالية والذي من المحتمل أن يعالج المشكلة ويوسع استخدام الحوسبة السحابية. ان حماية بيانات المستخدمين في السحابة مجالا بحثيا نشطا وقد أحرز تقدم كبير في تأمين بيانات العملاء في السحابة في السنوات القليلة الماضية. ومع ذلك، فإن البحوث الحالية تتعلق في الغالب بالهجمات التقليدية التي لا تستهدف التهديد الجديد الناشئ (أي تأمين البيانات من مشغلي الخدمات السحابية وغيرهم من المستخدمين في السحابة) أو تفتقر إلى التطبيق العملي في بيئة السحابة المتعددة أو تعاني من نقاط ضعف أمنية او اداء ضعيف يجعلها غير عملية. في هذه الأطروحة، نقترح الحلول القائمة علي FPGA لحماية بيانات المستخدمين من مشغلي السحابة وغيرها من أنواع مختلفة من الهجمات. الحلول المقترحة هي مناسبة لطبيعة العمليات في السحابة وهي عملية من حيث التكلفة والأداء. يمكن للحلول المقترحة في هذه الرسالة أن تستهدف مجالين رئيسيين هما: (1) تأمين البيانات الحساسة التي يملكها العميل الذي يقوم بإجراء العمليات على بياناته في السحابة، (2) تأمين البيانات الحساسة التي يتم تجميعها من مصادر متعددة ومعالجتها في السحابة مثل بيانات إنترنت الأشياء التي يتم جمعها من أجهزة مختلفة. واقترحنا طريقة آمنة لتجميع ومعالجة هذه البيانات في السحابة وفصلنا برامجها و تفاصيل تنفيذ الطريقة في FPGA وأظهرت النتائج أن الحلول المقترحة تتكامل بشكل جيد مع موارد السحابة الأخرى ويمكن أن تبدأ أسرع ب 15 مرة مقارنة بالبرمجيات الافتراضية التقليدية متوسطة الحجم على نفس السحابة وأداءها يمكن مقارنته مع معالجة البيانات الغير مشفرة. ولتحسين معالجة بيانات إنترنت الاشياء في السحابة، أظهرت النتائج أيضا أن حلنا المقترخ فعال من حيث استهلاك الموارد والاداء..

CHAPTER

INTRODUCTION

1.1 Motivation

Cloud computing is an emerging paradigm that has many benefits for users and enterprises. Reduction of capital costs, which is one of the essential benefits of cloud computing, makes cloud computing the ultimate choice for enterprises. However, cloud security is a major concern that makes cloud computing not appropriate for applications with sensitive data such as financial data processing, medical data and sensitive internet of things (IoT) data. Existing solutions either focus on protecting users' data against external or peer attacks only or lack a more robust attack model. There is an implied assumption that the cloud operator is a trusted entity. This leads many organizations with sensitive data not to process such data in the cloud.

Current cloud infrastructures are not fully secured since the cloud provider has access to users’ data on the cloud servers. According to ESG Insider Threats Survey [1], insider attacks, which is carried by a staff in the cloud company, was ranked at third most dangerous attacks of the cloud. Also, 66% of all organizations are very vulnerable to insider attacks methods [1]. Furthermore, 53% of respondents of US State of Cybercrime Survey confessed that damages caused by the insider attacks affect their business more than outsider attacks [2]. For example, Ristenpart et al. outlines how a malicious insider can extract RSA and AES keys in Amazon's cloud by exploiting shared caches [3]. Another case occurred at Twitter when many companies documents were revealed by Twitter administrator's account that was hacked by a malicious insider [4]. Therefore, there is a need for an effective solution that could build the trust between cloud service providers and the clients so that enterprises take advantage of the cloud to reduce their capital cost and economies of scale.

1.2 Problem Definition and Dissertation Contributions

The dissertation addresses the problem of protecting sensitive data processing in the cloud. The challenge of the problem is that the sensitive data need to be processed in a hardware resource owned by the cloud such that no one, even the hardware resource owner (i.e. the cloud provider), can disclose it while processing. The client outsources the sensitive data to be processed by hardware resources owned by the cloud, such as field-programmable gate arrays (FPGAs), and under the cloud premises, uses software provided by the cloud to authenticate the hardware resource, securely sends the FPGA application that is owned (partially or totally) by the client or another party to the cloud and securely outsource the sensitive data. Securing sensitive data processing in the cloud is even more challenging when the data is collected from multiple sources (i.e. IoT devices) that are deployed in locations under the premises of some party and are owned by the client or another party.

In this dissertation, FPGAs are utilized to secure sensitive data processing in the cloud. FPGAs can be integrated with other cloud HW resources to form flexible, scalable, independent and secure compute resources within the cloud infrastructure. Therefore, clients can safely perform the computation of their sensitive data in the cloud in a secure manner while utilizing the benefits of the cloud and the fast and secure computation of the FPGAs. Sensitive data can be farmed out from the untrusted cloud servers to FPGAs, which are configured by the client's application, for secure processing. Compared to conventional software-based systems, the attack surface is substantially smaller and better defined. This is because FPGA configuration does not require the involvement of operating systems, drivers or compilers, making them suitable to build security solution under more robust attack models and stronger security guarantees. Further, FPGAs can build more sophisticated solutions for modern machine-to-machine communication, IoT data processing and big data applications [5]. As utilizing FPGAs for data protection in the cloud is either limited or unsecured in the literature, there is a substantial need for an efficient and secure FPGA schemes to protect sensitive data in the cloud. Other CPU based attempts to solve this problem are not fully secure, not suitable for on-cloud IoT data protection and suffer from large overhead that make them impractical for medium and big data secure processing.

Hence, the dissertation has the following main contributions to address the problem of securely processing sensitive data in the cloud:

- An efficient and practical FPGA-based scheme for securing client sensitive data processing in the cloud from various kinds of attacks (including malicious cloud providers) which has a very little area overhead and can be efficiently integrated with other cloud resources.

- A scheme for protecting third party's intellectual properties (IPs) in the cloud. The scheme facilitates the use of IPs from third parties in the client applications who is not necessarily a hardware expert.
- A security scheme for securing IoT sensitive data processing in the cloud and a symmetric proxy re-encryption scheme for IoT data on-cloud transformation. The scheme is suitable for publish/subscribe systems. It was evaluated and a complete FPGA prototype for the scheme and the proxy re-encryption is presented in this dissertation.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows:

Chapter two presents a background in topics related to the contributions of this dissertation including cloud computing architectures, attack models and insider attacks in the cloud, an overview on trust in modern platforms, physically unclonable functions and proxy re-encryption. Chapter three reviews cloud computing security directions; including protecting users' data from other tenants and protecting users' data from the cloud provider. Chapter three also provides a literature review on the current research and products of trusted computing and secure processors. Chapter four presents an overview of the proposed security scheme for securing client data in the cloud. It also covers the FPGA implementation details and performance evaluation of our scheme. Chapter five covers the proposed scheme for IoT data protection in the cloud along with a symmetric proxy re-encryption to provide secure data transformation in the cloud environment. It also describes the existing IoT business models and presents the experimental results of our proposed cloud-integrated IoT scheme. Chapter five also discusses the performance results of the software and hardware implementations of our symmetric proxy re-encryption. Finally, the dissertation is concluded in chapter six.

CHAPTER

BACKGROUND

In this chapter, we give an overview of architectures and management aspects in recent cloud platforms. We will then discuss attack models and demonstrate how a malicious insider could utilize these architectures to carry out attacks to users' virtual machines and data. Further, we will provide an overview of topics that are related to this dissertation such as trusted computing, physically unclonable functions and proxy re-encryption.

2.1 Overview of Cloud Architectures and Management

The architecture of a cloud computing can roughly be categorized into four layers: the application layer, the platform layer, the virtualization layer and the hardware layer as shown in Figure 1.

The application layer: this layer is at the top of the hierarchy and consists of cloud applications. Cloud applications have the interesting characteristics of availability, lower operating cost compared to conventional applications, and automatic-scaling feature, which maintains applications' availability and allows an application to scale its capacity up and down to satisfy its needs.

The platform layer: operating systems and software framework lie in this layer. The goal of this layer is to make the deployment of the applications into the virtual machines simpler. For example, Microsoft Azure works at the platform layer to provide support for storage, and database for applications in the application layer.

The virtualization layer: Also known as the infrastructure layer, assigns computing resources and storage to the target virtual machine by dividing the hardware resources using virtualization technologies (called hypervisors) such as Xen [6] and KVM [7]. The virtualization layer is an important layer in the cloud computing architecture because it involves many recent topics related to the overall design of the cloud, such as dynamic resource allocation. Indeed, this layer is essential in terms of security of the cloud. As it is just above the hardware layer, any security solution would be brought down to the hardware layer.

The hardware layer: the hardware layer is responsible for managing the hardware resources of the cloud, such as physical servers, power, cooling systems, routers, and switches. This layer is implemented as what is called data centers in practice. The data center consists of thousands of servers organized in racks. These racks are interconnected through switches and routers.

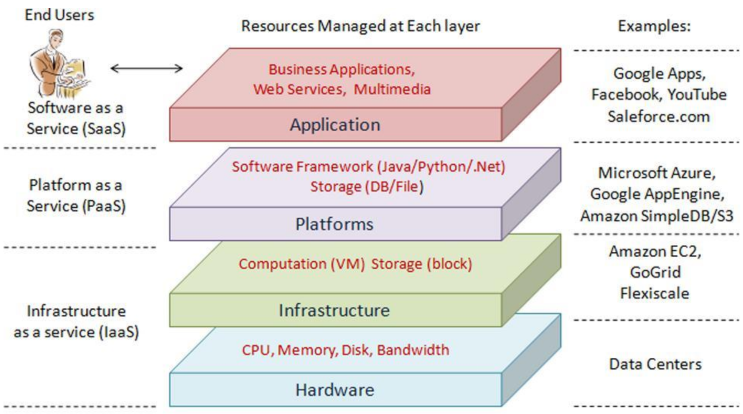


Figure 1: Cloud computing architecture [191]

Cloud computing architecture is modular and every architectural layer is loosely coupled with the layers above or below. This modularity allows layers to change separately similar to the design of the OSI model for network protocols. Conceptually, each layer can be seen as a client of the layer below and each layer can be implemented as a service to the above layer. Though, clouds offer services, in practice, they are gathered into three categories: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). IaaS denotes the provisioning of resources infrastructure. PaaS is for providing operating systems and software development frameworks. SaaS provides applications over the Internet. Users interact with the cloud using the web interface, which shows the SaaS. Then, the requests from the users are processed and deployed by PaaS and IaaS [8]. The business model of cloud computing is shown in Figure 2.

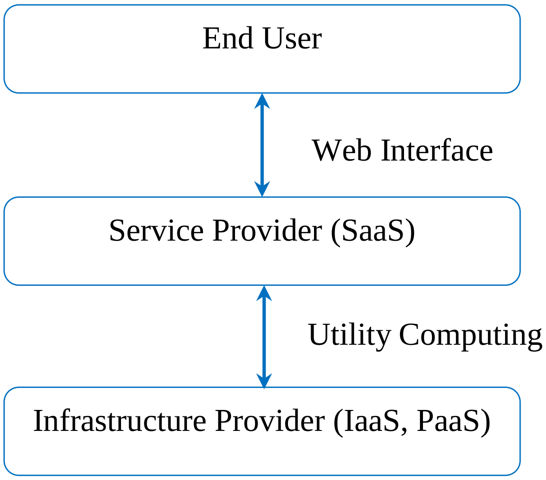


Figure 2: Business model of cloud computing

Mostly, the job of cloud providers is to manage the hardware resources, i.e. compute, network and storage resources that form the infrastructure of the cloud. Management of these resources can be classified into seven areas [9]:

- Global scheduling of virtualized resources: refers to the system-wide monitoring of virtual resources and the underline hardware resources consistent with cloud providers goals.
- Resource utilization estimation: This is necessary for both the cloud provider and the client because it increases the utilization of the physical and virtualized resources.
- Resource pricing and profit maximization: This is due to the nature of pricing used in the cloud. As the resource consumption is decreased, the cost is also decreased and vice versa. Hence, resource pricing needs to be managed in a proper way.
- Local scheduling of cloud resources: This type of scheduling is similar to operating system scheduler. It deals with one server, hosting multiple virtual machines, and schedules requests to the physical resources of the server such as CPU and memory.
- Application scaling and provisioning: To increase the performance of the service for large applications.
- Workload management: This is due to the fact the cloud user might be a business enterprise and hence the workload request from the enterprise users have to meet certain requirements from cloud user.
- Cloud management systems: For feedback of resources to the cloud user.

2.2 Overview of Attack Models and Insider Attacks in the Cloud

This section discusses attack models, which are classified into cryptographic attacks, network attacks and physical attacks, and cloud insider attacks as well as cloud cryptographic algorithms.

2.2.1 Cryptographic Attacks

Known-plaintext attack (KPA) is a type of attack which assumes that the attacker has the ciphertext and at least a limited number of samples of the corresponding plaintext. An example of such attack is the use of the encrypted bitstream and the unencrypted bitstream to extract the key embedded in the Xilinx FPGAs [10].

In Chosen-plaintext attack (CPA), the attacker specifies an arbitrary input plaintext and forces the encryption engine to produce the resulted ciphertext. Using the plaintext and the corresponding ciphertext, the attacker can infer information about the encryption algorithm and the key used for encryption. As an example, consider a file storage system that uses the same key to encrypt/decrypt users' files. The attacker can encrypt a file and obtain the corresponding encrypted file. Using these files, the attacker can extract the key used for encryption and use it to decrypt other users' files.

There are two types of chosen-plaintext attack, adaptive chosen-plaintext attack and batch chosen-plaintext attack. In the adaptive chosen-plaintext attack, the attacker can encrypt more plaintexts and obtain the ciphertexts of these plaintexts. The attacker has the opportunity to analyze the previous pairs before choosing a new plaintext as an input to the encryption engine. In batch chosen-plaintext attack, the attacker encrypts all of the plaintext before obtaining any ciphertext.

To differentiate between KPA and CPA, if the attacker is able to obtain a pair of plaintext and ciphertext, but not any specific pairs, then the attack is KPA and if the attacker is required to give input plaintext and obtain the corresponding ciphertext, then the attack is CPA. Note that KPA is a special kind of CPA.

Ciphertext-only attack (COA) is a type of attacks in which it is assumed that the attacker can obtain the ciphertext only and the plaintext is not accessible by the attacker. COA can happen most likely in real life cryptanalysis. However, it is the weakest attack because the attacker lacks information and only the ciphertext is available for him. Therefore, COA is typically the hardest to design and at the same time the easiest to implement. Exhaustive key search or brute force attack is an example of COA, in which all keys are tried by the attacker until the correct key is found. The success of COA depends on the length of the key and does not depend only on the cipher strength or the type of the cipher being used.

Chosen-ciphertext attack (CCA) is the opposite of CPA and the attacker chooses a ciphertext and obtains the corresponding plaintexts. This enables the attacker to investigate different portions of the plaintext state space and may enable him to find vulnerabilities. Types of CCA include lunchtime attack and adaptive chosen-ciphertext attack. The attacker in lunchtime attack have access to a limited ciphertexts and plaintexts pairs. The attacker in adaptive chosen-ciphertext attack can pick a sequence of ciphertexts to be decrypted and obtains the ciphertexts. For further steps, he has the chance to learn from the previous results and choose more effective next ciphertexts.

The open key attack model assumes that the attacker has some knowledge about the key used for encryption/decryption. Related-key attack is an attack of such model and the attacker uses a key that is related mathematically to the target key to encrypt the plaintext and access the ciphertext. Known-key distinguishing attack is another type of open key model and the attacker chooses a key and study the cipher and distinguishes between a random data and a ciphertext.

2.2.2 Network Attacks

Network attacks include wiretapping, port scan, idle scan, man-in-the-middle, impersonation, replay, ARP poisoning, ping flood, ping of death and Smurf attacks.

Wiretapping is a clear example of network attacks and is effective when the data sent is unencrypted. In port scan attack, the attacker sends packets to the victim machine while varying the port in every packet in attempt to know which ports are open and to identify the operating system and the services in the victim machine. Idle scan attack is a kind of port scan attack on TCP ports and the attacker sends packets to TCP ports using impersonated machines (i.e. machines with their identity stolen) to identify and learn the services on the victim machine.

ARP (address resolution protocol) is used to identify the MAC address of the target machine. The sender machine broadcasts a message over the network containing the IP address of the target machine. The target machine sends a reply containing its MAC address. ARP poisoning attack occurs when the attacker replaces the MAC address in the reply message by his own MAC address; causing the sender machine to send the traffic through the attacker machine. This attack is possible in local area networks that utilize ARP and can be used to launch other attacks such as denial of service attacks [11].

Ping flood is a type of denial of service attack in which an attacker sends ping packets to the victim machine as fast as possible without waiting for replies. The attack is more effective when the bandwidth of the attacker is more than the bandwidth of the victim. Ping of death happens when the attacker sends a ping packet that is larger than the bytes allowed by the IP protocol. The packet gets fragmented, sent and reassembled in the victim machine. When reassembled, buffer overflow occurs which causes system crash and allows injection of malicious code [12].

Internet Control Message Protocol (ICMP) is part of the internet protocol suite and is used to send control messages such as error and ping messages. The attacker in Smurf attack spoofs the IP address of the victim machine and broadcasts large ICMP messages to the network using this IP address. If number of machines in the network is large, the responses from these machines will flood the victim machine; causing denial of service [13].

2.2.3 Physical Attacks

Physical attacks are divided into invasive, non-invasive and semi-invasive attacks [14]. Invasive attacks are hardware attacks that require manipulation of the physical properties of the chip. Non-invasive attacks are similar to invasive attacks but do not damage the chip package. Semi-invasive attacks are relatively new type of attacks that require depackaging the chip, similar to invasive attacks, but do not create contacts with internal chip lines.

Invasive attacks require expensive equipment, knowledge and time. A well-known attack of this type is miroprobing; where a needle is attached onto the internal wires of the chip to extract the chip secrets. Non-invasive attacks require moderate level of equipment and knowledge to implement. Non-invasive attacks include side-channel, brute force, fault injection and data remanence attacks [15].

Side-channel attack is any attack that use data about the encryption or decryption process to break the system such as using the noise created by encryption engines or measuring the time of various computations. Side channel attack includes, generally, cache, timing, power analysis and electromagnetic attacks. Cache attack is based on monitoring cache accesses made by the user in a shared environment such as cloud servers. Timing attack is based on measuring the time it takes to do the operation such as measuring how many cycles a memory access take to identify whether the access is read or write. Power analysis attack makes use of power consumption that is varied by the hardware during computation. Power analysis attacks is classified into simple power analysis (SPA) and differential power analysis (DPA). SPA obtains the information directly from the power consumption of the device (current versus time), while DPA obtains the information from power variations by observing differences between traces of different operations and statistical analysis is applied to obtain the secrets from noisy measurements that are difficult to analyze using SPA. Electromagnetic attack is based on electromagnetic radiation, which can provide information about plaintexts and may directly disclose the plaintext and other information. Fault injection can be used to exploit the erroneous result or unexpected behavior of the chip to extract its secrets. Data remanence is the sensitive data that is read by the attacker but is supposed to be erased [15].

Semi-invasive attacks use ultraviolet (UV) light, X-rays and other light sources to disturb chip operations and extract sensitive information. Attacks of this type include UV, optical fault injection, and optical side-channel Attack [16].

There are a number of attacks that are targeting FPGAs such as Reverse Engineering, Tampering, Cloning, Counterfeiting, and Crippling attacks. Reverse Engineering can cause bypassing security measures of the configuration. An adversary can study the configuration blocks and replace security components by his own malicious components in order to disclose configuration secrets and sensitive data. Tampering is a special type of reverse engineering, where the adversary modifies the configuration to gain access to its secret keys or interrupt its functionality or disclose its data. Hardware Trojans are a clear example of tampering [17]. Cloning attack occurs when an exact copy of the FPGA configuration is created by an adversary. Counterfeiting attack is an extension to the cloning attack and it occurs when all FPGAs of the same family and size are identical. Thus, a configuration made for one device can be used with another. This can be easily done in the cloud environment [18]. The details of the design do not need to be known by the attacker and the configuration is just regarded as a black-box reducing the effort of compromising the FPGA to insert a snooping circuitry to disclose FPGA secrets and therefore disclose users' data. Crippling attacks are similar to denial of service attacks on networked servers. The attacker re-configures the FPGA with an invalid configuration to bring the FPGA system offline [19].

2.2.4 Insider Attacks in the Cloud

How a malicious insider (MI), who could be an employee in the cloud, could carry attacks to users data of the cloud was reported by many survey publications, such as [20][21]. An MI could utilize the system to carry out various kinds of attacks depending on his position in the cloud. An administrator who can manage client's virtual machines (VMs), for instance, can do anything to the VM he is managing. Similarly, an employee working in cloud hypervisors might inject a code to monitor all the activities of client's VMs on the underlying hardware [20]. Memory Dump Scanning, Templates Poisoning and Snapshot Cracking are examples of exploiting client's information in the cloud [21].

Dumping the memory to get sensitive information is called Memory Dump Scanning attack; the MI can easily dump the memory to a flash or external storage. Although the retrieved data would be large and sensitive data is concealed in hundreds of megabytes of data, the MI usually use techniques such as social engineering, which count on human communication and involves tricking them into breaking security measures, to extract the critical data.

Cloud providers usually provide templates for the virtual machines images to be created from. The default templates and virtual machines that are deployed from an infected template could be downloaded and analyzed by an employee; passively revealing VM owners' sensitive data. This attack is relevant to many platforms and is also applicable to OpenStack which is the top open source cloud computing technology in 2014 [22].

Cloud providers manage the users and passwords of the VMs in a normal manner, assuming trust of their staff. Not all VMs disks are encrypted due to many difficulties facing disk encryption such as sharing resources with other tenants [23]. If an MI is a VM administrator, he can easily make an attack on that VM by simply reading all the required information from the unencrypted information.

2.2.5 Cryptographic Algorithms for Cloud Computing

This section reviews cryptographic algorithms used or preferred in the cloud. According to Soofi et al. [24], most approaches are based on RSA encryption. 60% out of the 30% RSA encryption techniques results are validated by experiments. Figure 3 below depicts the use of encryption algorithms in the cloud. Patwal and Mittal [25] also reported that RSA, DES and AES are widely implemented in the cloud.

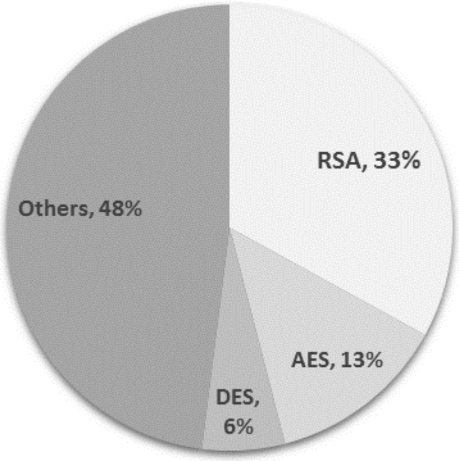


Figure 3: Percentage of use of various encryption algorithms in the cloud

An algorithm called Diffie–Hellman–Merkle key exchange (DH) was invented by Ralph Merkle for sharing a key between two parties over unsecured channels [26], [27]. The secret key can then be used as a session key to encrypt/decrypt the data the two parties want to send/receive. The algorithm works as follows:

Let the two parties be A and B.

A and B agree on a prime number p and base g (which is normally a small number such as 2).

A chooses an integer a and sends B g^a *mod p.*

B chooses an integer b and sends A g^b *mod p.*

A computes $K = (g^b \bmod p)^a \bmod p$

B computes $K = (g^a \bmod p)^b \bmod p$

A and B now share the secret K.

All values except a and b are non-secret and can be sent between the two parties. For example, if A and B agrees on p to be 23 and g to be 5, then A chooses a = 6 and sends B the value 8 and B chooses b = 15 and sends A the value 19. K then will be 19^6 mod 23 $=8^{15}$ mod 23 = 2 which is the secret they now share. Ephemeral Diffie-Hellman can be used such that every time A and B share a secret, different shared key is created enabling what is called perfect forward secrecy, which means that even if the private keys are compromised, past communications are still secure.

Leading cloud computing platforms use the DH such as OpenStack [28], which uses this algorithm for key sharing between its components. The DH is one of the best protocols of sharing keys between parties and in this dissertation, we make use of its basic principle for key sharing between FPGAs and other parties because of its lightweight computation which is suitable for cloud computing and IoT.

2.3 Trusted Computing

The Trusted Computing Group had developed the Trusted Computing (TC) technology. TC is an attempt to ensure that computers will behave as expected and this behavior would be enforced by hardware and software. The enforcement is achieved by including a special chip integrated with computers' hardware, which includes unique, inaccessible by other components of the system, encryption keys. The concept of trusted computing leads to the fact that the hardware of the system is theoretically secure from all kinds of attacks, including its owner [29].

Trusted computing is implemented in practice as a hardware component attached to other hardware components of computer assets, in addition to software drivers. Trusted computing witnessed a remarkable success in personal computers. Unfortunately, it is in its early stages in cloud computing because the intention of trusted computing was not targeting virtualization [30]. The most common and most widely used hardware component is the trusted platform module (TPM). Hence, we will consider, in the following subsections, the TPM in our discussion and will discuss the implications of using the TPM in the cloud paradigm.

2.3.1 Trusted Platform Module

The Trusted Platform Module (TPM) is a special chip issued by the Trusted Computing Group (TCG) to secure hardware by embedding cryptographic keys into devices. The TPM was developed to provide device identification, authentication, measurement, encryption, and device integrity. Software can use the TPM to authenticate hardware devices and the TPM is capable of monitoring and reporting configuration state by using the main bus of the computer from the point of computer power-up.

A TPM has at least 16 Platform Configuration Registers (PCR registers), which are initialized to a known value when the machine is rebooted. The values of these registers cannot be arbitrarily set. The values of the PCR registers can be retrieved from the TPM by issuing the TPM Quote operation.

The TPM is mainly used to create a foundation of trust of the software installed in the host where the device is present. A process called Static Root of Trust for Measurement

(SRTM) performs a chain of measurements, starting when the host platform is reset, of the components and configuration data involved in the system boot while the Dynamic Root of Trust Measurement (DRTM) is the process of taking the measurements while the system is running. Each component measures the next component before passing the control to it, forming what is called a Chain of Trust (CoT). The CoT, at least, involves the BIOS, the boot loader and the operating system kernel. The resulting measurements must be always the same unless the boot components are modified. The combination of the TPM Quote operation and the SRTM process, allows the remote attestation of the host [32]. An external attester can request a TPM Quote of the PCRs, and compare the obtained values with a baseline of the PCR values of the system generated when it was in a trusted state.

There are three keys produced by the TPM; Endorsement Key (EK), Storage Root Key (SRK) and Attestation Identity Keys (AIK). The Endorsement Key is created by the TPM manufacturer and is never released outside of the TPM. EK is used to ensure that the data was encrypted by the TPM (the data can be trusted). A private EK which is used to encrypt the data can be proven to be from the TPM by using it to decrypt a value that has been encrypted with the public EK. When the TPM is initialized by the user, in the process of taking the TPM ownership, the SRK is generated. It is used to protect TPM keys created by applications, so that these keys cannot be used without the TPM (all the keys the user requests are produced by the SRK). Finally, the AIK are used as an alias of the EK for signing information produced by the TPM, e.g. the PCR register values issued after the TPM Quote operation. The problem with the TPM, in general, is the unsecured bus between it and other computer components and most of the attacks are actually carried out from this unsecure bus [33].

2.3.2 Intel Trusted Execution Technology

Intel's Trusted Execution Technology [34] is a technology developed to provide attestation of a platform and the operating system running in that platform and to ensure that the OS starts in a trusted environment. TXT relies on the TPM and another chip to provide the measurements of the platform components and the software. TXT aim is to provide an isolated environment for VMs from untrusted software by providing the VM control over the platform while it is active. A warm system reset is performed by the authenticated code module (SINIT ACM) before starting the VM. The TPM measures the hardware and updates the SRTM registers and when the VM is initialized, it updates the DRTM registers. Therefore, the VM can start from a valid state and can execute in a trusted environment. Similar to the TPMs, TXT is vulnerable to physical DRAM attacks since DRAM encryption is not implemented. The System Management Mode (SMM) is the most privileged execution mode in which all executions are suspended and only a special software stored inside a firmware (stored in a ROM and is theoretically inaccessible by the operating system or applications) is executed. Several attacks were reported to the SMM such as [35]–[39], which leads to granting an attacker access to all computer software including accessing the TXT memory.

2.3.3 Intel Software Guard Extensions (SGX)

Intel SGX follows the approach in Figure 4 below. Data owner has the measurement of the initial state as he computes its hash locally by creating the exact container (or enclave) in his local machine. The attestation key (the private part of it) is used to sign the hash M, g^A and g^B and it is decrypted by the data owner by the AK public key, which is publicly available and is certified using the Endorsement Certificate (only the public key can verify information signed with the private key). The hash is done for the initial state only and any other code or data that will be received later will not be measured. After successfully attesting the container and sharing K using Diffie-Hellman key exchange, the data owner sends the code and the data encrypted to the enclave using K.

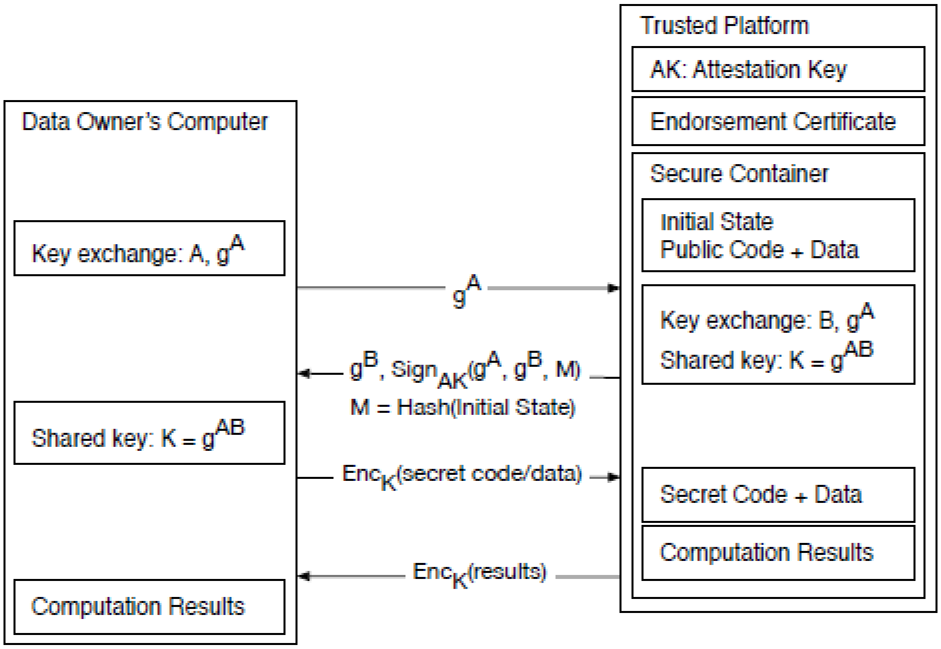


Figure 4: Trusted platform proves to a remote data owner that it is communicating with the right container [40]

The Data owner creates an enclave in his local machine using ECREATE instruction, Figure 5. ECREATE instruction will result in reserving part of the virtual address space for the enclave, creating an isolated part in the Processor Reserved Memory (PRM) called enclave page cache (EPC) and its Metadata (EPCM). EPCM is used to store information about the enclave and its pages and to ensure that each EPC page belongs to exactly one enclave.

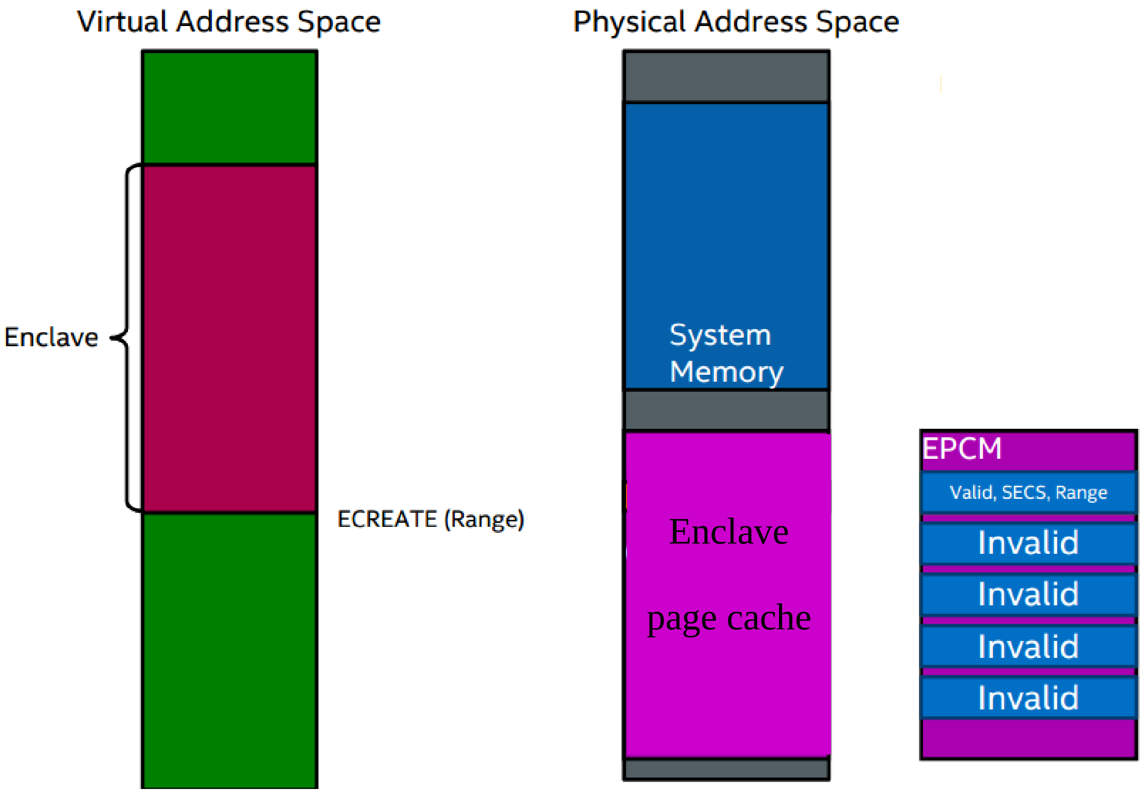


Figure 5: Creating an enclave [41]

Data owner then executes EADD instruction, Figure 6, to copy the required pages to the enclave area one by one. MRECNLAVE is a measurement register to store the measurement of the hash. EEXTEND instruction updates the enclave’s measurement

(MRECNLAVE) used in the software attestation process. The SGX Enclave Control Structure (SECS) is inside the EPC and stores enclave metadata and is used by the CPU for the enclave identity.

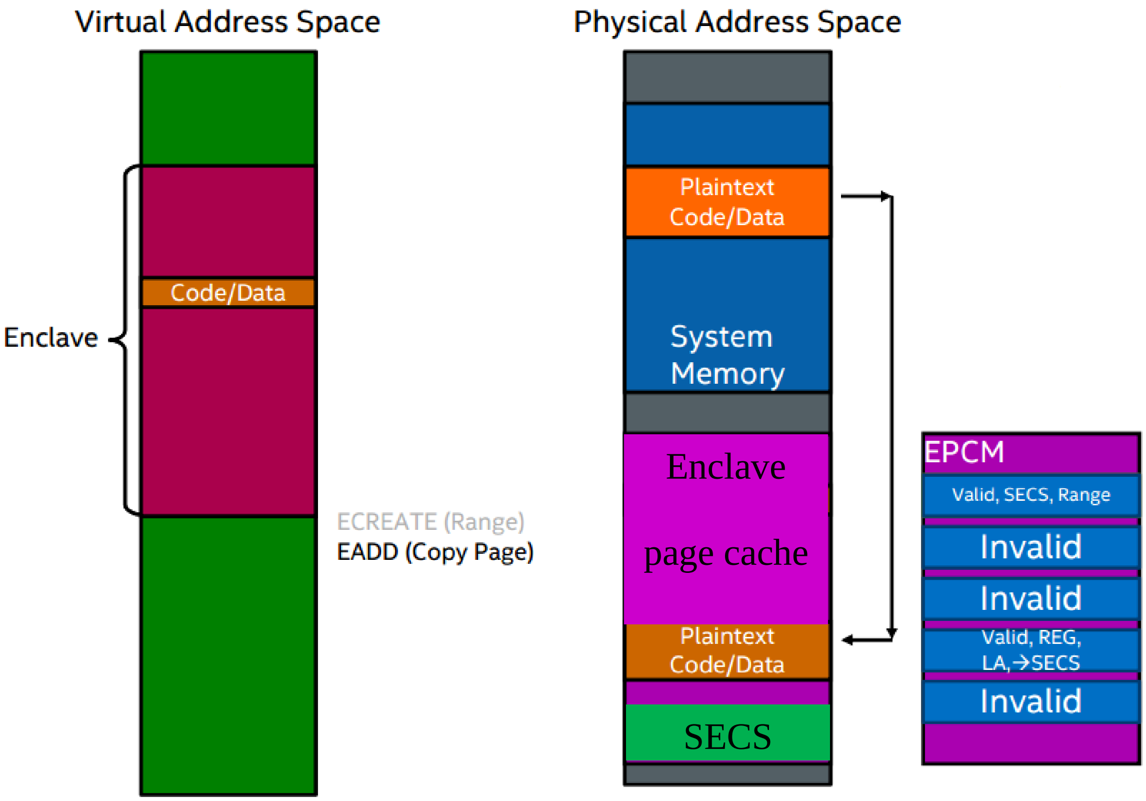


Figure 6: Copying enclave pages to the EPC [41]

Other instructions are shown in Figure 7. The EINIT instruction marks the enclave's SECS as initialized. Setting INIT to true means that EADD cannot be invoked on that enclave anymore and MRENCLAVE will hold the final measurement. The EENTER instruction is used to execute enclave code. The EEXIT instruction is used when the enclave code finishes performing its task to return the execution control to the process which invoked the enclave.

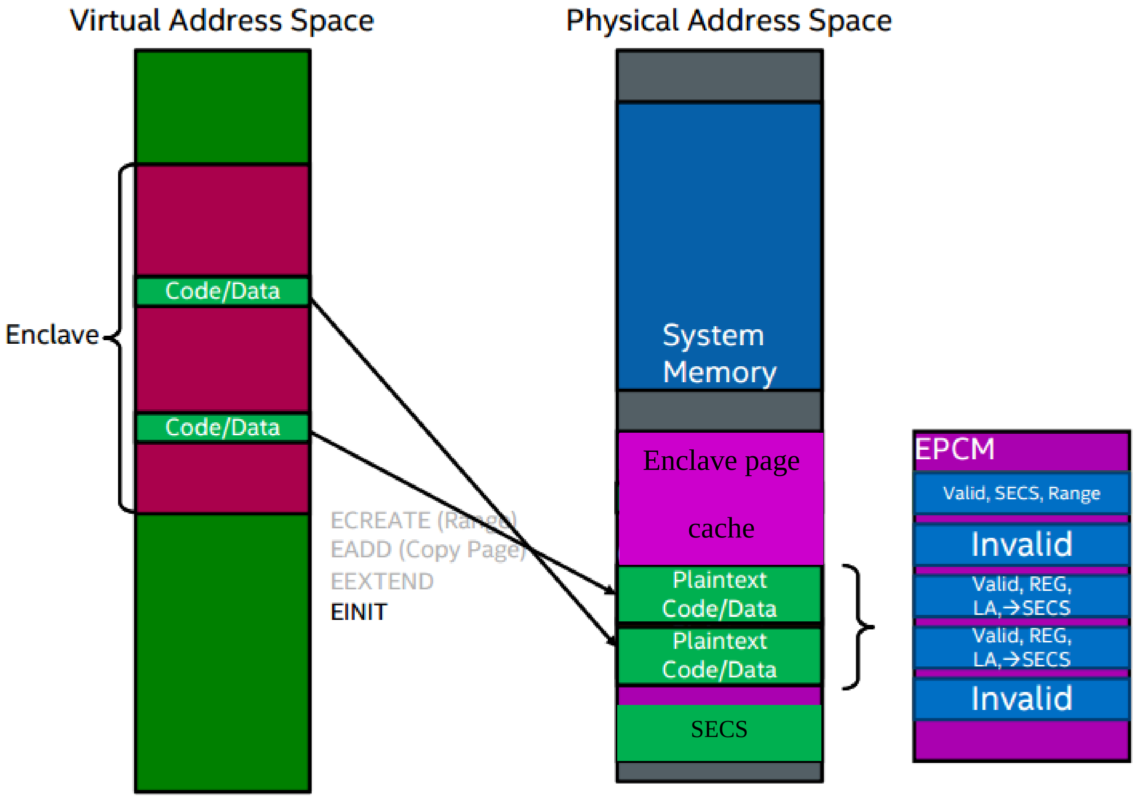


Figure 7: Enclave instructions [41]

The code and data are unencrypted inside the CPU package and if the code/data leaves the CPU package, they are encrypted using a symmetric key randomly generated on power up (Figure 8).

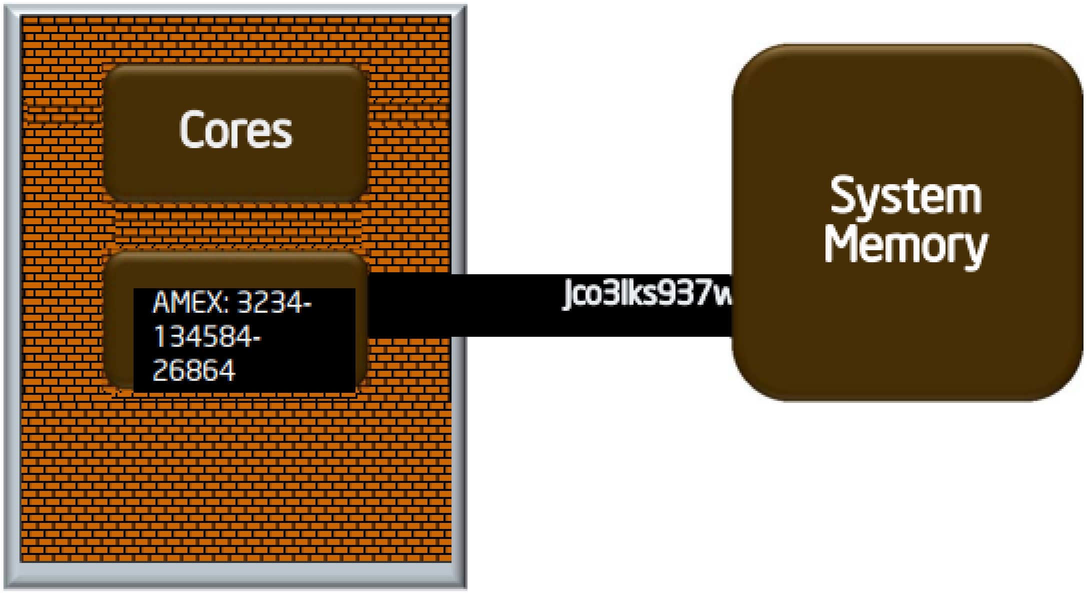


Figure 8: Code/data are encrypted outside the CPU package [41]

To run the enclave in a remote host, the same procedures described above are done in the remote host. Hence, the same value of MRENCLAVE will be produce by the remote host. Data owner can remotely attest MRENCLAVE value and compare it with his own value. As a result of the remote attestation process (as in Figure 4), a symmetric key can be shared and this key can be used to send encrypted data to/from the enclave.

2.4 Physically Unclonable Functions

Physically unclonable functions (PUFs) topic is one of the hottest topics in the field of chip authentication and reconfigurable computing. PUFs are functions that make use of the manufacturing process variations to uniquely identify devices. These variations cannot be controlled and therefore making PUFs behavior and response unpredictable. Figure 9 shows how a PUF can be used for device authentication. An ideal PUF, when a challenge is applied, should have a response that is unique (a device response should not match a response from another device), steady (when the same challenge is applied the device should give the same response), random (is based on uncontrolled variations and cannot be predicted) and tamper resistance (when the PUF is propped, it gives a different response).

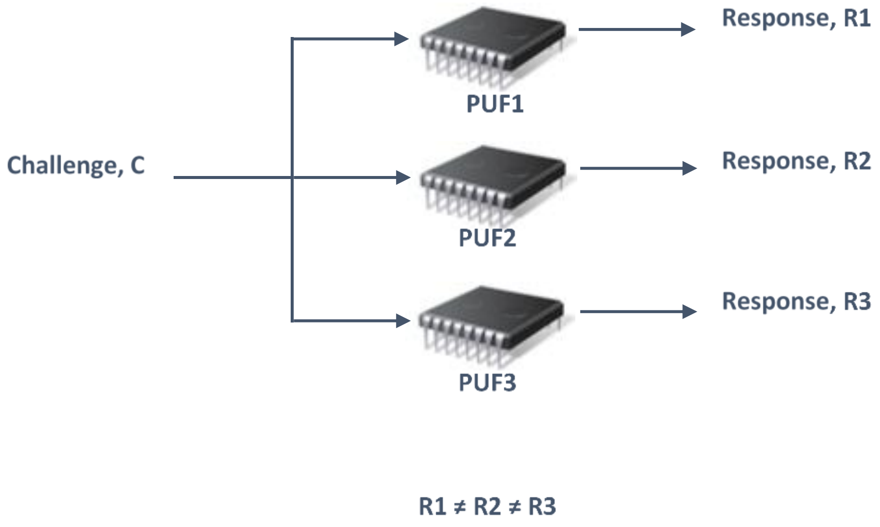


Figure 9: PUFs challenge response

PUFs can be classified into two categories; PUFs using explicitly-introduced randomness and PUFs using intrinsic randomness. Optical PUF and Coating PUF are two different types of PUFs using explicitly-introduced randomness. Optical PUF was one of the first attempts in producing unique identifiers for integrated circuits (ICs) and is formed when a transparent material is doped and when a laser beam is induced to the material at certain angle, a unique and random pattern is formed. This pattern is very difficult to duplicate and therefore this type of PUFs is unclonable. Optical PUF is rather laborious because it requires laser setup and a complex mechanical system to position the laser beam [42]. On the other side, Coating PUF is built when a network of wires is created on the top layer of an IC. The space between the wires is filled with a material and doped with dielectric particles randomly. Therefore, the capacitance between these wires would be random; creating a unique identifier for the device [43].

PUFs using intrinsic randomness do not require modifications to the manufacturing process and therefore are highly attractive. PUFs of this type can be divided into a delay-based PUFs and memory-based PUFs. Delay-based PUFs include Arbiter PUF [44], Ring oscillator PUF [45] and Glitch PUF [46] and memory-based PUFs include Butterfly PUF [47] and SRAM PUF [48]. Arbiter PUF is based on two parallel racing paths with an arbiter at the end of the two paths. A signal is applied simultaneously to the two paths and the resulting analog delay difference is processed by the arbiter to get the required digital value. Ring oscillator PUF utilize the delay characteristics of multiple ring oscillators and a comparison of them is made to produce the digital value. SRAM PUF and Butterfly PUF are called memory-based PUFs. SRAM PUF [49] makes use of the initial values of the SRAM cells as these initial values are different from chip to chip due to the variations in the manufacturing processes, in which the state of the bit at power up determines its initial value. Butterfly PUF is an emulation of SRAM PUF in FPGAs.

PUFs in FPGAs include Arbiter PUF [44], Ring oscillator PUF [45]; enhanced by many other publications such as [50]–[53], Anderson PUF proposed by J.H. Anderson [54], Butterfly PUF [47], and SRAM PUF. Anderson PUF is considered as Glitch PUF and is based on the delay of lookup tables (LUTs) in shift register mode and carry chain multiplexers in the FPGA slices to extract the physical variations of FPGA devices.

SRAM PUF was made possible using recent Xilinx [55] and Altera devices [56]. In this dissertation, we assume the use of the SRAM PUF for generating random numbers as other PUFs are currently suffering from low stability, uniqueness, and entropy compared to SRAM PUF [57].

Although many types of the PUFs can be implemented using most of nowadays FPGAs, we believe that most of these PUFs are still facing many drawbacks when used in FPGAs and we though recommend the use of SRAM PUFs as it is considered more appropriate for FPGAs and are already available in recent FPGA devices [56].

2.5 Proxy Re-encryption

Proxy re-encryption (PRE) is a method for transforming a ciphertext c1 encrypted using a key (K1) to another ciphertext c2, which can be decrypted using a key (K2), without decrypting/disclosing the plaintext during the transformation. The benefit from such transformation is that the time for transformation is less compared to decrypt-then-encrypt and the transformation does not expose the plaintext, which means that the entity that does the transformation (i.e. the proxy) is not necessarily trusted. PRE was first introduced by Mambo and Okamoto [58] and later Blaze, Bleumer, and Strauss [59] proposed the BBS scheme in which a semi-trusted proxy converts a ciphertext encrypted by user's A public key to another ciphertext to be decrypted by user's B private key. Semi-trusted in this context means that the proxy will correctly execute the code required for the conversion and does not mean that users A and B trust it partially. The conversion is possible when user A provides a re-encryption key to be used for the conversion. The

BBS was improved by Ateniese et al. [60] and many other public key based PREs were proposed such as [61] [62] [63] [64]. These public key based PRE methods are used when the delegator (user A) is for some reason unwilling to receive the data and want to give delegation to another party (user B) to decrypt the data without revealing his private key to user b or the semi-trusted proxy. For example, a manager wishes to give delegation for his employee to check his email messages without giving him his private key. The manager gives a re-encryption key to the email server so that the messages are converted by the email server and his employee can decrypt the data without fully trusting the email server or giving the employee the manger private key. Symmetric key based PRE scheme was proposed by Syalim et al. [65]. The scheme uses the all-or-nothing transform [66] which means that the ciphertext cannot be decrypting if any part of it is missing. However, the scheme requires the generation of 8 keys to do the conversion and these keys had to be shared between the two communicating parties and the proxy; making the conversion complicated. The preferred properties of the PRE are:

1- Unidirectional: a PRE scheme is unidirectional if the proxy is able to transfer delegator ciphertext to the delegatee ciphertext but not the opposite.

2- Non-transitive: The proxy cannot give the delegation alone to a third user C from the re-encryption keys of users A and B. for example, user A gives re-encryption key (rK1) to delegate the rights to user B and user B gives re-encryption key (rK2) to delegate the rights to user C. Then the proxy cannot give the delegation rights of user A to user C by using rK1 and rK2.

3- Collusion-resistant: if the proxy and another user collude, they cannot obtain the private key of the other party.

4- Non-interactive: when the private key of the delegatee is not required to create the re-encryption key, then the PRE is non-interactive.

5- Multi-use/Single-use: if only the original ciphertext can be re-encrypted, then the PRE is single-use and the opposite is correct for multi-use.

6- Key-privacy: if the identity of the delegator and the delegatee cannot be identified by the proxy from the re-encryption keys and the ciphertext, then the PRE is key-private.

7- Transparent: a PRE is transparent if the delegatee is unable to note the existence of the proxy between him and the delegator.

8- Key-optimal: the PRE is key-optimal if it is required to store small amount of data (the private keys only).

9- Temporary: if the delegator can delete the decryption rights after some time, then the PRE has the temporary property.

CHAPTER

LITERATURE REVIEW

In this chapter, we will discuss recent contributions related to securing users' data in the cloud. This includes surveying the techniques published in securing users' data from both other cloud clients and the cloud providers. We will also investigate recent advances in trusted computing as it is a fundamental topic in outsourced data secure processing. Finally, we explore secure processors and their implementation methods.

3.1 Protecting Users' Data from Other Users

Protecting users' data from other cloud users requires securing user's virtual machines form other virtual machines in the same cloud server. There are two lines of research heavily investigated in the literature; securing hypervisors from being compromised and virtual machine isolation.

Ferrie [67] described an attack scenario for identifying which hypervisor is being used to deploy the virtual machines. Leading hypervisors (QEMU, VirtualPC, Bochs, Parallels, Hydra and VMWare) were compromised using his scenario. The attack is based on specific unusual instructions that leads to raising exception that are not handled by the hypervisors. The work also provided a solution for each hypervisor by handling the unusual instructions. Similar attacks were also reported by Joanna [68] and Tobias [69].

Louis and Jordan discussed some of the vulnerabilities related to granting access to users' metadata in hypervisors [70]. Among these attacks, redirecting data flows using firewall Ports, utilizing user application interface, hooking calls to library and hooking system calls were illustrated in detail.

Breaking the isolation between virtual machines is also carried out on the weak parts of the hypervisor. This attack causes Denial of Service, System halt, and memory overflow. Attacks similar to this one were reported in many contributions in the literature such as [71]–[73].

Fog computing and decoy methods were used intensively to protect users' data in the cloud. Fog computing relies on making distributed cloud resources such that the cloud parts (storage and servers) can be geographically closer to the user. Fog computing was mentioned to be more secure because it places the data geographically close to the end user and therefore certain policies can be applied to restrict access to cloud resources to users in the same geographic area [74]–[77]. Decoy files/documents are trap files that are used to trap illegitimate user. The search behavior of an illegitimate user is random to some extent when entering the system and when the trap file is hit, it will fire an alarm. If the alarm is a false positive, the legitimate user will be challenged with a question and his legitimacy will be checked.

Several attempts were carried out to tackle the attacks to users' data from other users using Fog computing and Decoy method and these techniques are not helpful for tackling attacks from the providers of the cloud since the architecture of the cloud itself is being ignored as well as dumping the memory of cloud servers. Most previous studies are theoretical and are not directly related to commercial clouds. A common factor between all of the attacks discussed in this section is that attackers utilize a weakness in the software such as the hypervisor. Therefore, these types of attacks targeting other virtual machines in the cloud are not new and are not specific to the cloud; unlike attacks from cloud staff. Attackers utilize vulnerabilities to attack users' data and cloud providers close the vulnerabilities and this process continues until the probability of finding a vulnerability is close to zero. Many software attempts exist in the literature targeting these kinds of externally considered attacks which obviously assumes that cloud providers are trusted [78]–[83].

Table 1 summarizes the existing SW approaches for protecting data in the cloud. All approaches are not secure against cloud insider attacks and completely ignore attacks that read memory of the cloud servers such as reading the cryptographic keys of the client.

Table 1: Summery of SW approaches targeting data protection in the cloud

Approach	Publication	Year	HW	Consider cloud insider	Secure against physical attacks	Cryptographically secure
Fog Computing	[84][85][77] [76]	2012-2016	×	×	×	-
Decoy files	[74]–[76]	2014	×	×	×	-
Neural networks	[86]–[88]	2013-2014	×	✓	×	×
others	[32], [89]–[92]	2013-2016	×	✓	×	×

3.2 Protecting Users' Data from Cloud Providers

Homomorphic encryption is the process of performing computation on ciphertext and the encrypted result when decrypted matches the expected result of the computation performed on plaintext [93]. Homomorphic encryption could be the solution to the problem of securing clients' data in the cloud. However, to date, no homomorphic encryption algorithm that is computationally achievable has been developed [94]. From the cloud perspective, scalability of the solution is a major feature in the cloud and using homomorphic encryption systems with their computation cost would contradict this feature [95]. Furthermore, most of the new homomorphic encryption algorithms such as ad hoc polynomial approaches and bilinear pairing are not fully addressed and examined by the cryptography community when compared to other popular algorithms such as RSA and AES [95].

To protect circuit IPs (intellectual properties) of third parties and prevent cloning, FPGA vendors provide a symmetric key embedded on the FPGA to secure bitstreams. Clients use the vendor's SW tool to encrypt their bitstreams (i.e. FPGA configuration) which would then be decrypted inside the FPGA. Data could also be included or initialized within the bitstream which makes it secure. After the data is processed, it can be readback as part of the bitstream (which is encrypted by the FPGA before it is outputted). Again, the vendor's SW would decrypt the data for the client. There are many problems with this approach. First, these keys are only made available by the FPGA vendors to major clients only under very rigorous qualification requirements. Second, large data cannot be initialized in the FPGA due to limited space resources. Third, the configuration process is extremely slower than the FPGA's processing speed which means that almost all of the time would be spent on transferring data in and out of the FPGA, thus wasting the FPGA's processing power. Finally, and most importantly, many successful attacks against such a method have been reported, either using Known-plaintext attack (KPA) [10] or other types of attack [96]–[104].

Eguro and Venkatesan proposed an FPFA-based security approach for cloud computing [95]. This approach implements the security components as Static Logic (i.e. fixed, non-reconfigurable macros) inside the FPGA and makes use of RSA and its private key to form a root of trust (ROT) inside the FPGA. A certificate authority (CA) would certify the public and private keys for every FPGA. The private key is a crucial component from the security point of view because the FPGA is within the cloud and the key could be obtained by reverse engineering the Static Logic or inserting a sniffing circuitry to sniff the private key since the RSA-based solution did not consider checking the integrity of the Static logic.

Some researchers proposed providing trust for FPGA-based embedded systems by implementing a full Trusted Platform Module (TPM) [105] on the FPGA [106][107]. The use of TPMs assumes a secure channel and requires users to take ownership of the TPM and set a chain-of-trust. This is not only impractical in a multi-tenant cloud environment (where each user would need to take ownership of the TPM and set their own chain-of-trust), but does not guard against Man-in-the-Middle (MiM) and replay attacks by a malicious cloud administrator. Analysis of virtualized TPMs (vTPMs) showed that they are less secure than physical TPMs (pTPMs) [108], even with a trusted host (i.e. the cloud). vTPMs have the same vulnerabilities to attacks by malicious cloud administrators as pTPMs.

In [33], researchers proposed a data security model for users data that is similar to TPMs. They proposed augmenting the cloud's servers with two types of chips that are paired (cryptographically) by the manufacturer; an FPGA as a processing chip and an ASIC as state chip that holds the state between power cycles (using non-volatile memory). The system guarantees integrity and freshness by maintaining a Merkle tree, which is a tree of hash values and the value of each node depends on the values of its child nodes, for user's data in the cloud storage. However, no implementation was provided nor a performance evaluation. Furthermore, it is assumed that the FPGA is 'securely' packaged and that there is a secure channel between the state chip and the certificate authority. In addition, there were no key management policies for different users nor guarantees of integrity of the configured circuitry. Similar to the conventional pTPMs, this model is not suitable for the multi-tenancy nature of the cloud and have similar vulnerabilities to attacks by malicious cloud administrators.

A framework for users' data privacy (mainly for Map-Reduce applications) in the cloud using the security features of current FPGAs and proxy re-encryption was proposed in [109]. It uses public/private keys for encrypting/decrypting a symmetric key to be shared between the user and the FPGAs in the cloud for the purpose of data encryption/decryption. It also uses the FPGA's embedded symmetric keys for bitstream protection, which is not only not available for general users, but was already proven to be insecure against KPA as stated earlier. In addition, the proposed scheme requires a CA to certify FPGAs public keys as well as a proxy server to manage key re-encryption. The scheme assumes full trust in the cloud user (who will get access to the FPGA's symmetric keys) and semi-trust in the proxy and cloud operator. This is another major drawback of this scheme. Further, it is not clear why a proxy re-encryption was needed since it does not seem to have a real impact on the security of this framework. Specifically, a proxy is not needed since both the FPGA and the user have their own public/private keys and if there is a need for scheduling more than one FPGA for the user, the cloud provider can just send the public key of each FPGA (that is certified by the CA) directly to the user.

3.3 Trusted Computing in the Cloud

This section discusses the state-of-the-art in establishing trust in the cloud and reports the complexity of such establishment. It also describes Intel's Software Guard Extensions (SGX) and its role in establishing trusted computing environment in remote hosts such as the cloud servers.

Eisenbarth et al. proposed a reconfigurable architecture with TPMs [106]. The approach allows for scaling and updating the TPM functionalities and including it into the chain of trust which makes it possible to bind sensitive data to the reconfigurable hardware. This work used a fixed logic bitstream for enabling the root of trust and the FPGA boots from this bitstream. However, the problem of using partial reconfiguration to integrate the bitstream of the client with the fixed logic bitstream was not discussed.

Developing a mechanism for the user to attest the state of the host by remotely attesting the TPM PCR registers was presented by Kekkonen et al. [110]. However, sharing the TPM among applications used by different users in the same platform was not addressed.

The problem of extending trust to the cloud is an active area of research. Several publications attempted to propose trusted models for the cloud. In the rest of this subsection, we summarize these attempts and give our conclusions regarding trust in cloud computing environment.

Santos et al. proposed the design of a trusted cloud computing platform (TCCP) that uses the TPM to provide trusted hardware [111]. However, the proposed platform was based on the TPM which is meant to be used per device and not per virtual machine. Another attempt to use the TPM in the cloud was addressed by Neisse et al. This work relies on

TPM to provide trusted computing, which is not suitable for multi-tenant nature of the cloud as stated earlier [112].

The virtualization of the TPM to bring its security properties to virtual environments is not direct due to its design and security constraints. This issue was discussed by Cucurull and Guasch [108] and an overview of virtualizing TPM in Xen QEMU virtualization solutions was provided. The idea of virtual TPM in Xen is simply an emulation of the TPM and there is a manger that control these emulations while QEMU uses the passthrough technique to achieve virtualization of the TPM. Passthrough is a simple method that passes all PCR registers of the TPM to the VM. Similar emulation was proposed by Bertholon et al. [113].

The possibility of using TPM to establish trust in Cloud Computing, between the cloud provider and the customers was also studied by Achemlal et al. [114]. Similar work was carried out by Berger and Caceres [115] and what they have proposed for virtualizing the TPM is shown in Figure 10. The disadvantage of this architecture is that TPM information sent to client VMs pass through the hypervisor which exposes the information to insider attacks. The proposed scheme by Berger and Caceres propagates the idea of the TPM to the software level which reduces the success of such methods. There are many similar attempts to establish trust in cloud computing including [116]–[119].

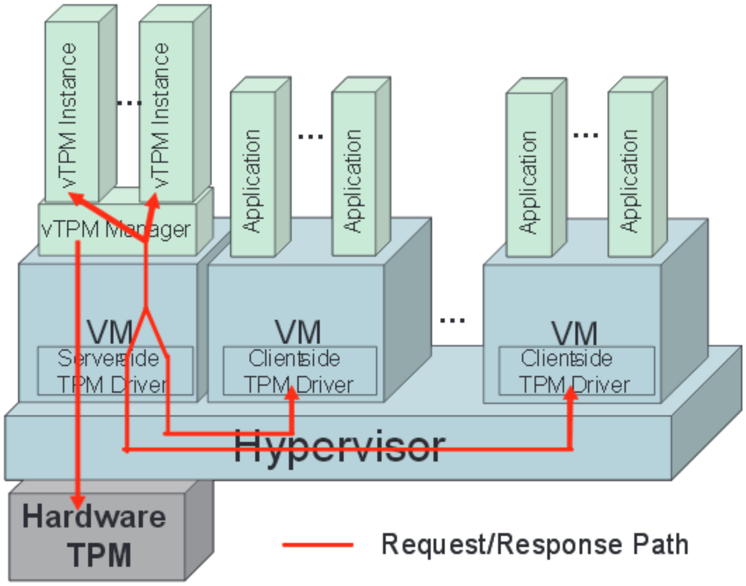


Figure 10: Virtual TPM Architecture of [115].

Zic et al. [120] and Nepal et al. [116] proposed the Trusted Extension Device (TED) to enable mobile trust. Thilakanathan et al. used it to build the root of trust in the cloud [117] although one user was assumed for the TED to be integrated with the cloud servers, Figure 11.

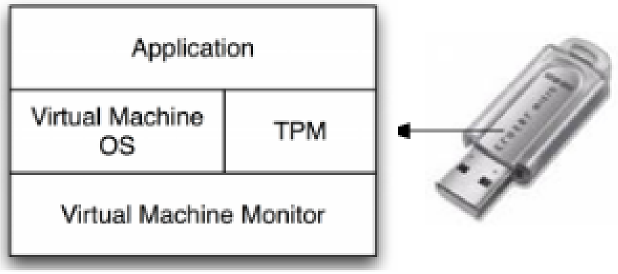


Figure 11: The abstracted architecture of the trusted extension device (TED)

TPM offers a robust way for providing trust in single machines although there are several successful attacks reported in the literature. However, TPM is not suitable in its current design for cloud computing due to the fact that TPM role has to be extended to the software level which contradicts the goal of the TPM.

3.3.1 Intel Software Guard Extensions

Recently, Intel announced the Software Guard Extensions (SGX) [121], which is a set of instructions to extend Intel architecture. SGX instructions aim to provide security-sensitive computation integrity and confidentiality guarantees where privileged software such as the operating system (OS), the kernel and hypervisors are untrusted. To provide integrity and confidentiality, SGX allows user-level code to allocate private regions of memory, called enclaves, that is protected from other processes; including processes running at higher privilege levels. SGX also provides software attestation, which is the process of proving to the user that his code is running in the intended trusted hardware. The TPM attestation covers all the software in the platform while the TXT attestation, discussed in Section 2.3.2, covers the VM code. SGX attestation covers only the private code and data inside an enclave. SGX does not prevent side-channel attacks such as Cache-timing attacks, Physical attacks and Microcode attacks [122]. Successful cache attacks were reported by many researchers including [123]–[126]. The problem of such attacks in the cloud is that an enclave has the power to control the server and other users in the same server would be compromised; spreading the attack to insiders and outsiders. Moreover, Intel SGX requires modification to the application using it which is not practical in all cases [127][40].

3.4 Secure Processors

In addition to Intel SGX, several secure processors were proposed in the literature. This section summarizes secure processors that are relevant to the topic of trusted computing.

The Aegis secure processor [128] depends on a security kernel, which is a subset of the OS kernel, to isolate containers from each other by configuring the page tables used in address translation. The security kernel handles processes, virtual memory management, and hardware exceptions and uses processor's features to protect itself from other components such as untrusted device drivers. Aegis' security kernel is assumed to be a trusted part of the OS and it can evict the containers pages while verifying the paging operations correctness. This means that the OS can learn the memory access patterns of the container. Further, cache timing attacks can be carried on Aegis.

The use of a trusted hypervisor to provide secure applications running untrusted systems was presented in the Bastion architecture [129]. The Bastion architecture allows the hypervisor to enforce the container to use specific memory locations by maintaining a Module State Table that stores a page table containing each physical memory page and its container and virtual address. The hypervisor checks that the virtual address used to map a physical memory page matches the virtual address associated with the physical address of this physical memory page in the Module State Table. Similar to other secure processors, the Bastion hypervisor is vulnerable to cache timing attacks and untrusted operating system can evict the container pages; allowing it to learn a container's memory accesses.

Sanctum [130] partition the DRAM into equal continuous regions to isolate the container and each DRAM region is allocated to exactly one container. Flush on context switches is performed to isolate the containers in the caches. Sanctum relies on a trusted security monitor, similar to the SMM, to ensure that the container can only reference to a memory inside its DRAM partition. The container manages its page table and also handle its page faults, which means that the OS cannot learn the virtual address causing the page fault. The Sanctum design does not protect against any physical attack and focuses on software attacks. Also, Sanctum does not prevent fault-injection attacks and timing attacks.

The Ascend [131] and Phantom [132] secure processors make use of the oblivious RAM. Oblivious RAM is a concept that performs the reads and writes simultaneously to hide the operation being performed and shuffles the RAM contents from time to time to avoid observing the patterns of accessing the memory locations. It follows that Ascend and Phantom do not suffer from attacks that probe the DRAM address bus and other attacks that attempts to learn memory access patterns of the containers. However, they incur large slowdown compared to other processors. It is worthwhile to mention that unlike SGX, which uses the Enhanced Privacy ID (EPID) [133] to preserve the privacy of the user using the SGX in a remote host, these secure processors did not guarantee the privacy of the user. The user can be tracked by the identity of the processor he is using.

Table 2 depicts existing secure processors and their characteristics. These processors are not suitable for IoT data processing because they target securing the data and the code coming from one user, which is not the case in the IoT paradigm as the data is coming from multiple sources (i.e. IoT devices) and is processed by the code sent by the user.

Table 2: Summery of existing secure processors

Processor	Publication	Year	Secure against timing attacks	Considers privacy of the client	Secure against physical attacks	Require modification to the OS
SGX	[121]	2015	×	✓	×	×
Aegis	[128]	2003	×	×	×	✓
Bastion	[129]	2010	×	×	×	modification to the hypervisor
Sanctum	[130]	2016	×	×	×	×
Ascend	[131]	2012	✓	×	×	✓
Phantom	[132]	2013	✓	×	×	✓

3.5 Summary and Discussion

This section summarizes the findings of surveying the existing methods for securing users' data in the cloud. Existing work in the literature is related to protecting users' data from other users in the cloud and from external attacks while only few contributions considered protecting data from the cloud providers. In addition, most of them consider normal software approaches towards solving the problem which are not suitable in the absence of trusted computing in the cloud.

For protecting users' data from other users in the cloud, all the surveyed and possibly existing techniques are based on software approaches. The directions for these techniques include virtual machine isolation techniques and making cloud hypervisors more secure. In addition, Decoy and Fog computing methods were intensively published for this kind of protection. Without doubts, these directions completely ignore the problem of protecting users' data from cloud staff such as those who engineer and manage the hypervisors.

There are several attempts found in the literature to protect data from cloud providers. These methods were hardware-based. However, no real implementation was reported and the proposed methods lack many factors to be implemented, discussed in detail in previous sections. These attempts do not consider trusted computing for their proposed methods. Providing trust for FPGAs is necessary if the FPGA is going to be put in the cloud. There is no relevant work proposed regarding this direction and mostly because FPGAs are used offline by the FPGA owner. Now, with the existence of FPGAs in the cloud for security, it will be mandatory to consider trust in the FPGAs and maintain the integrity of client's applications running inside them.

In traditional cryptography, encrypted data should be decrypted first to be processed. Fully Homomorphic Encryption (FHE) is a special type of cryptography that allows arbitrarily computation on encrypted data. This type of cryptography would allow for arbitrary computations on the cloud and the ability to store all data encrypted and performs computations on encrypted data. Unfortunately, as of today, FHE is not practical and introduce serious performance issues that would eliminate the original advantages of adopting cloud computing.

Trusted computing in cloud computing is at its early stages. Trusted computing prevents devices owner from modifying the hardware of the devices which is exactly what the cloud needs to protect clients' data. The concept of the cloud gives the clients the rights to prevent cloud providers from controlling their data in the cloud hardware. Trusted computing fits well in personal devices while, up to now, it is not fitting well in the environment that is supposed to do (i.e. the cloud). Therefore, the concept of trusted computing should be the basis for developing new trusted models for the cloud (i.e. hardware approach as it is for the TPM). Though, trust computing in its current form is not be possible to emerge to the cloud because of the multi-tenant nature of the cloud. A new approach most probably would dominate. The problem of preventing cloud providers from disclosing users' data had to be tackled in hardware as introduced by Intel SGX. However, Intel SGX suffers from poor performance for medium/big data sizes and suffers from various kinds of attacks such as side-channel attacks.

In summary, Table 3 shows, in general, different approaches that can be utilized or proposed for securing data in the cloud. Most of these approaches are HW-based and all of them have vulnerabilities/ weaknesses that could lead to disclosing clients' data in the cloud. Further, it is obvious that these approaches are not suitable for IoT data processing in the cloud.

Table 3: Summery of existing approaches that can be utilized for data protection in the cloud

Approach	Scalable	Suite multi-tenant	HW	Consider cloud insider	Secure against physical attacks	Cryptographically secure	Weakness
SW	√	√	×	×	-	-	-
Homomorphic	×	√	×	√	√	√	expensive
TPM	×	×	√	×	×	√	Not stuitable for virtualized environment
TXT	×	√	√	√	×	√	Cloud insider can break the security oracle
FPGA-based [95]	×	×	√	√	×	×	Cloud insider can modify the static logic and can imersonate the FPGA
FPGA-based [109]	×	√	√	√	×	×	The security infrastructure of the static logic is not secure
SGX & other processors	√	√	√	√	×	√	Suffer from timing attacks and incure large overhead

CHAPTER

SECURING CLIENT DATA IN THE CLOUD USING FPGAs

This chapter discusses our FPGA-based scheme for securing users' data and applications in the cloud. This includes the protocols, and all the HW and SW components required to implement the proposed scheme. The chapter also discusses the benefits of the scheme such as perfect forward secrecy, FPGA authentication, a secure symmetric session key establishment between the on-cloud FPGA and the client, and user's configuration integrity check while running in the cloud FPGA. The details of the complete proof-of-concept prototype along with the cloud testbed for implementing the prototype is also provided; including resource utilization, Synopsys tool synthesis results area of about 0.0265 mm2 in a state-of-the-art 16/14 nm fabrication technology if implemented as custom circuits on the FPGA (i.e. Hard Macros), and the boot time the FPGA take for the client to start using his/her application.

4.1 Introduction

Cloud computing has proven to be of eminse benefits for individual users and enterprises. Reduction of capital costs, which is one of the essential benefits of cloud computing, makes cloud computing the ultimate choice for enterprises. However, protecting the integrity and privacy of users’ data is a major concern that hinders the adoption of cloud computing for applications with sensitive data such as financial data processing and medical data [134]. Existing solutions focus on protecting users' data against external or other users’ attacks only and they assume that the cloud provider is trusted. This leads many organizations with sensitive data not to process such data in the cloud [135]

In this chapter, we discuss the proposed scheme for securing client data which has the following features:

1. Practicality: The scheme does not use additional resources, other than the FPGA which is already deployed in the cloud [136], nor any special activities between the clients and the FPGA vendor.
2. Secure client data processing: the scheme provides strong protection against a wide range of attacks including MiM attack, FPGA impersonation, replay attacks, etc. The scheme does not consider the cloud operator as a trusted party and it ensures the integrity of the client’ applications. It provides the standard overall protection as outlined in [137].
3. Suits the multi-tenant nature of the cloud.

In Section 4.2, an overview of the proposed scheme is introduced including the protocol, its security analysis, and the related framework (HW and SW components). Experimental results are presented in Section 4.3. This includes the complete cloud testbed implementation and the details of all components, their implementation details and performance figures. It also provides performance comparisons with conventional virtual machine boot times and other approaches such as Intel SGX. Section 4.4 demonstrates how our scheme can be used to protect intellectual properties (IPs) in the cloud. Finally, conclusions are presented in Section 4.5.

4.2 The Proposed Scheme for Securing Client Data in the Cloud

This dissertation presents a novel scheme that utilizes FPGAs to secure users' data and applications in the cloud. The proposed scheme protects against various types of attacks, provides FPGA authentication, and ensures fixed logic and user's application integrity, data confidentiality and configuration integrity. Architectures for integrating FPGAs into the cloud to implement the proposed scheme have also been developed and a complete prototype was implemented to evaluate the performance of the proposed scheme. It should be noted that the scope of our proposed FPGA-based secure cloud computing is different from existing published work on integrating heterogonous resources with OpenStack, namely the SAVI test-bed [138]. Our goal is to use FPGAs in the cloud to secure the clients' data while SAVI test-bed aims to make FPGAs a resource for cloud providers to utilize. Our scheme is more closely related to the user's side while SAVI test-bed is more closely related to the provider's side. Our solution is also very scalable (paramount to any cloud-based computing), thus have great potential for secure cloud-based computing as discussed also by other researchers [139].

4.2.1 Preliminary

Assumptions: In this work, we assume that the on-cloud FPGA devices have the following capabilities that most of which already exist in current FPGAs:

- Uniquely identifiable using any public identifier such as a printed serial number or other means such as a unique device DNA, similar to that found in Xilinx FPGAs [140]. This nonvolatile, unchangeable and permanently programmed value can be used to authenticate FPGAs running client's configuration. However, device DNA alone is not suitable for device authentication as was illustrated in [141].
- External reconfiguration and readback ports are disabled [142]. External ports such as JTAG and SelectMAP are used to program FPGAs and to read back the configuration in its current state inside an FPGA for debugging purposes.
- Configurable through an internal configuration access port (ICAP) such as in Xilinx devices [142]. An ICAP receives the configuration bit stream from the Static Logic and partially configures another portion of the FPGA. Hence, the FPGA should also support partial reconfiguration,
- Supports readback of static configuration contents such as Look-Up-Tables, interconnects, and I/Os only, but cannot readback dynamic data such as RAM or Flip-Flop contents.
- Have standard high-speed communication ports such as 100 Gigabit Ethernet to enable their in-cloud usage.

4.2.2 FPGA Static Logic

We assume that, in the future, FPGA manufacturers would provide adequate support for securing users' data and applications on their FPGAs. Specifically, we assume that cloud-based FPGAs would have the necessary static or fixed logic in the form of hard macros (i.e. non-configurable static logic) that supports different security schemes. These macros include a PUF and masking circuitry to generate random numbers to be used by the modular exponentiation circuit for Key generation, modular exponentiation circuit, AES block for Encryption/Decryption, SHA3 for hashing the b, authenticating the FPGA and hashing the configuration readback that is used by the client to ensure that his/her application is not modified, and a controller to coordinate the different activities. Figure 12 shows these components and the connection between them and main state control is an FSM that controls the operation of these components. It should be noted that the proposed scheme could be implemented using current FPGAs that do not have the required Static Logic outlined above. The Static Logic in this case would be provided by the board manufacturer as pre-configured circuitry on FPGAs on tamper-proof boards and packages. Boards should be shipped with batteries and be powered constantly to maintain the Static Logic's configuration. Users' circuits, are placed into specific FPGA regions via partial reconfiguration. Only the input/output of the Encryption/Decryption would be made available to the users. This alternative solution allows anyone to make these boards and act as a trusted authority.

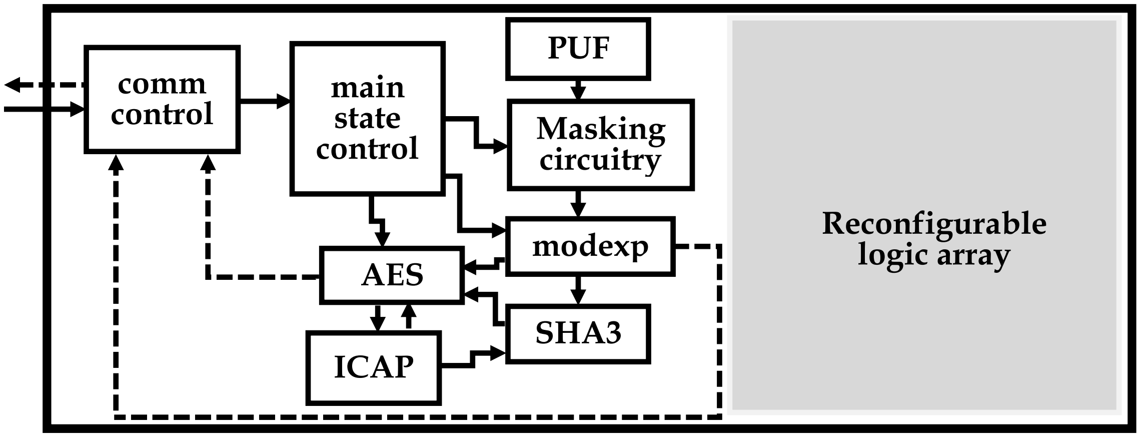


Figure 12: The proposed FPGA structure and the components of the static logic. Dotted lines represent outputs.

4.2.3 The Overall Framework

Notation: PUF-RN is the on-FPGA, n-bit, PUF-generated random number that is read once by the manufacturer and cannot be read again or altered. The session mask M, is also an n-bit random number generated by a trusted authority. M is used to generate an L-bit random number b from the PUF-RN. RN is a secure random number generated by the client. Bit(client) is the partial bit stream representation of a client's design. Config is the actual FPGA configuration obtained through readback using ICAP. Encryption of a message msg using a key k is denoted as MSG := Enc(msg,k) and the corresponding decryption as msg: =Dec(MSG,k). We use E_BIT(client) to denote encrypted bit streams: E_BIT(client) := Enc(Bit(client),k). F symbolizes an FPGA device. ID(F) is the identifier value and is used to uniquely refer to a specific FPGA.

Involved parties: Figure 13 shows the parties involved in our proposed scheme. In addition to the client and the Cloud Provider (CP) who is providing FPGA-based processing as a service, the FPGA Vendor (FV) who sells FPGA devices to cloud operators also acts as a trusted authority (TA). It is not necessary that the FPGA manufacturer be the TA. Alternatively, an OEM (board) manufacturer could act as the TA. In this case, it will get the PUF-RN from the manufacturer.

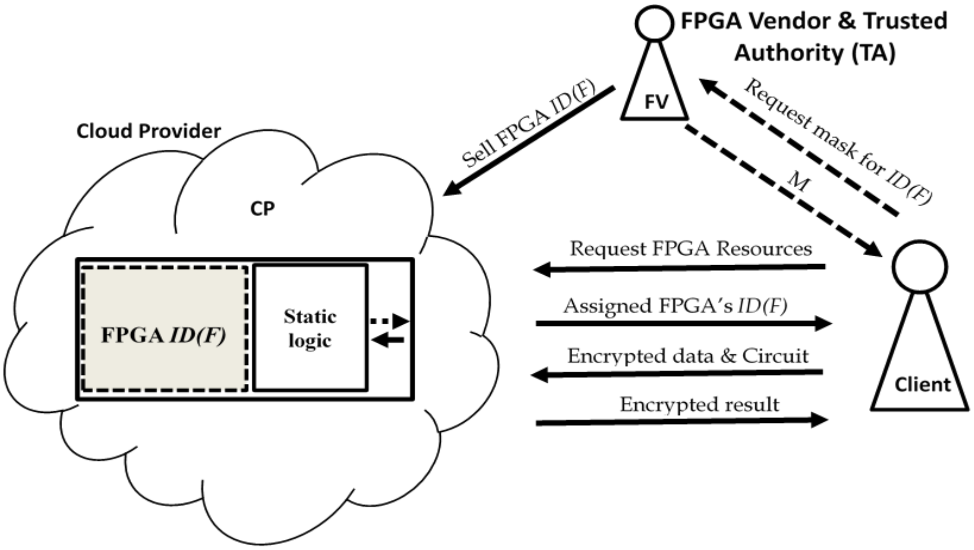


Figure 13: The proposed framework of the scheme.

4.2.4 The Security Protocol

The proposed 10-steps protocol for securing users' data on the cloud using FPGA processing is illustrated using the sequence diagram in Figure 14:

- The client sends a request for a physical resource (i.e. the FPGA) to the CP. The CP assigns an FPGA for the client and sends back its identifier (ID(Fi)),
- The client forwards the ID(Fi) to the TA, which has the value generated by the PUF and stored in the PUF-RN for each FPGA. The TA responds with the following FPGA authentication credentials; a session mask M (a random n-bit number with exactly L number of 1s), hash of the L-bit number b concatenated with ID(Fi), and the FPGA's session key portion g^b mod p. Note that both g and p

are public values with g usually being a small integer such as 2 and p being a prime number satisfying the conditions Length(b) ≥ length(p) and $g^b \geq p$,

- The client forwards M and its own portion of the session key, g^a mod p, to the CP and requests FPGA authentication credentials. Similarly, g^a must be ≥ p, and Length(a) ≥ length(p). The FPGA will use M to generate b using the masking circuitry described in Section 4.2.5. The FPGA uses b to generate its portion of the session key, g^b mod p, hash (b+ ID(Fi)), g^b mod p and sends the result back to the client. The client can now authenticate the FPGA by comparing the values of Hash (b+ID(Fi)) and g^b mod p received from the TA and CP. This prevents MiM and FPGA impersonation attacks [26]. Both parties now share the symmetric session key g^{ab} mod p, completing the Ephemeral Diffie–Hellman key exchange [26]. At this point *a* and *b* are destroyed by the client and the FPGA, respectively. In addition, the session key will be destroyed at the end of the session so as to achieve the desirable security feature of Perfect Forward Secrecy (PFS),
- The client encrypts his/her circuit's configuration bit stream Bit (client) using g^{ab} mod p, and sends it to the FPGA. The fixed logic on the FPGA will then decrypt it and use it to configure the FPGA through the ICAP,
- To protect against any circuit tampering (e.g. HW Trojans or sniffing circuitry inserted on the FPGA), the client chooses a secure random value RN, encrypts it with the session key g^{ab} mod p and sends it to the FPGA requesting configuration readback. The Static Logic decrypts RN, reads back the FPGA configuration, hashes it with RN, encrypts with the session key, and sends it back to the client. The client can use this to validate the integrity of the FPGA. This check can be

repeated any number of times (with a new RN every time to prevent replay attacks), during the operation of the client’s circuit on the FPGA.

The steps above are repeated for every session and M is never repeated. It should be noted that this scheme also supports 3rd-party provided circuit IPs (i.e. the circuit is provided by an IP vendor). In this case, to protect the circuit IPs, the IP vendor will encrypt the circuit IP(s) using a different Mask and key (obtained through similar steps), and perform the integrity checks.

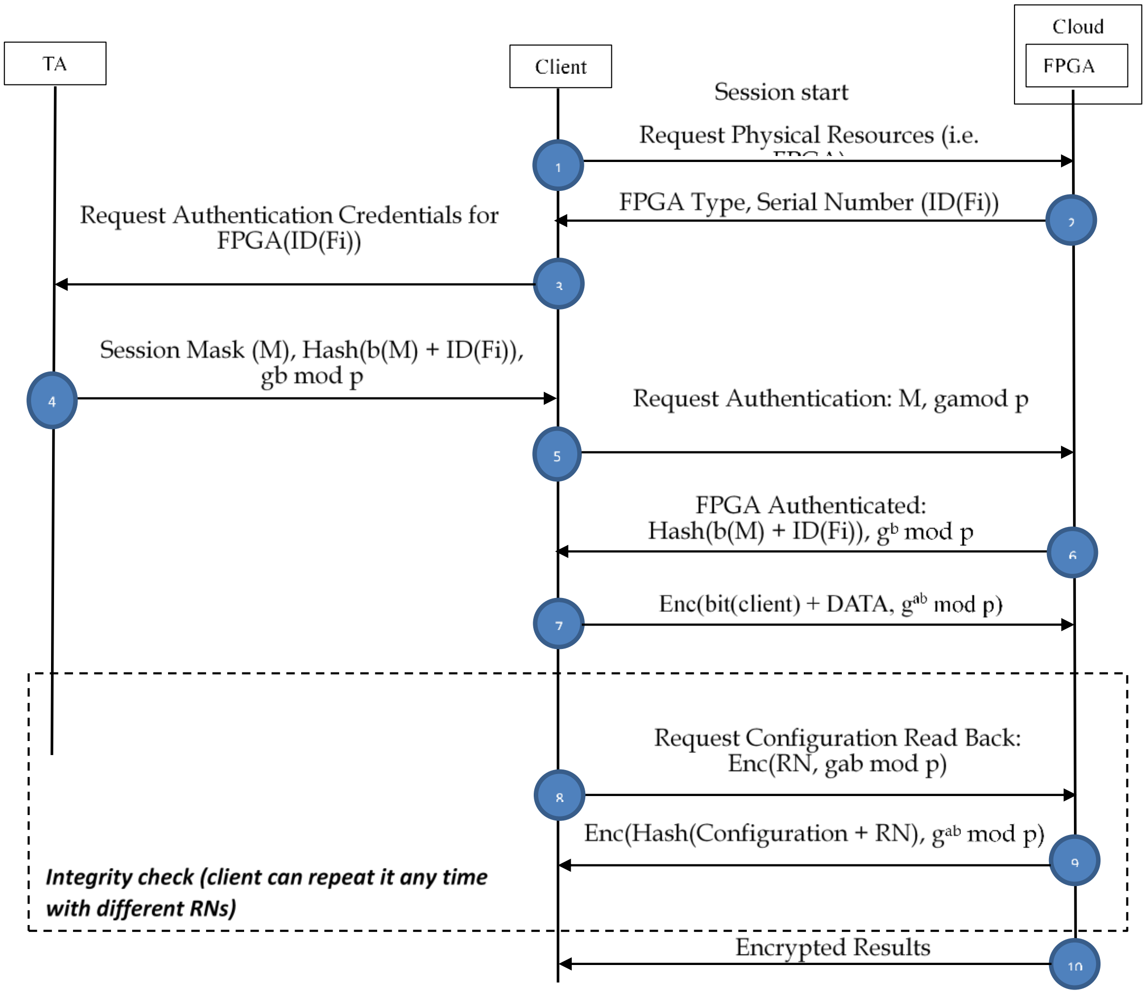

TA
Client
Cloud
FPGA
Session start
Request Physical Resources (i.e. FPGA)
1
FPGA Type, Serial Number (ID(Fi))
2
Request Authentication Credentials for FPGA(ID(Fi))
3
Session Mask (M), Hash(b(M) + ID(Fi)), gb mod p
4
Request Authentication: M, gamod p
5
FPGA Authenticated: Hash(b(M) + ID(Fi)), gb mod p
6
Enc(bit(client) + DATA, gab mod p)
7
Request Configuration Read Back: Enc(RN, gab mod p)
8
Enc(Hash(Configuration + RN), gab mod p)
9
Integrity check (client can repeat it any time with different RNs)
Encrypted Results
10

Figure 14: The protocol sequence diagram.

A slight variation on the protocol, Figure 15, could be made so that the TA does not need to compute g^b mod p while still providing strong protection against MiM attacks. In this case, the FPGA sends the double hash Hash(Hash(b(M) + ID(Fi)) + g^b mod p) along with g^b mod p to the client. The client then computes the same double hash (using Hash(b(M) + ID(Fi)) received from the TA and g^b mod p received from the FPGA) to authenticate the FPGA. This version of the protocol requires slightly more cycles to compute the double hash but relieve the TA from performing the modular exponentiation operation.

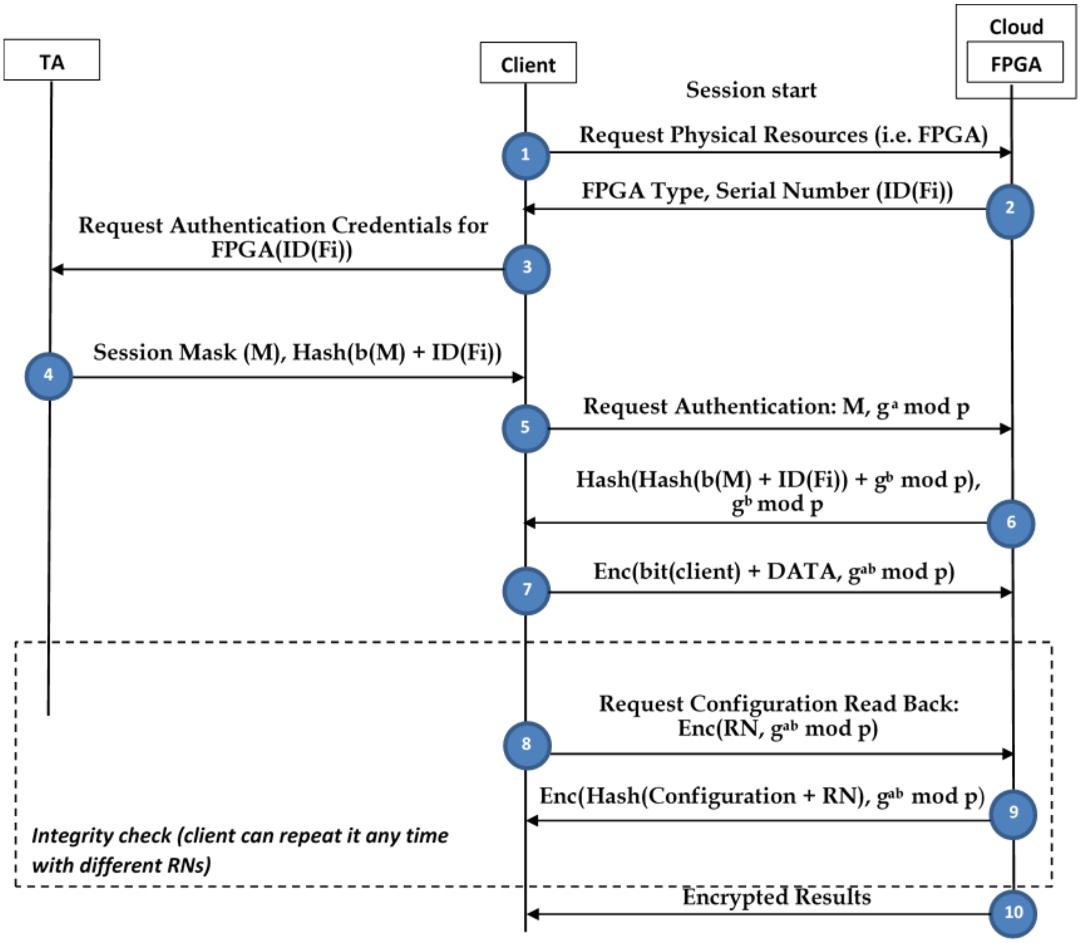

TA
Client
Cloud
FPGA
Session start
Request Physical Resources (i.e. FPGA)
1
FPGA Type, Serial Number (ID(Fi))
2
Request Authentication Credentials for FPGA(ID(Fi))
3
Session Mask (M), Hash(b(M) + ID(Fi))
4
Request Authentication: M, g^a mod p
5
Hash(Hash(b(M) + ID(Fi)) + g^b mod p), g^b mod p
6
Enc(bit(client) + DATA, g^{ab} mod p)
7
Request Configuration Read Back: Enc(RN, g^{ab} mod p)
8
Enc(Hash(Configuration + RN), g^{ab} mod p)
9
Integrity check (client can repeat it any time with different RNs)
Encrypted Results
10

Figure 15: A variant of the protocol. Double hashing is computed by the FPGA and the client to avoid sending g^b mod p by the TA.

4.2.5 The Masking Circuitry

The masking circuitry, as shown in Figure 16, consists of an n-bit PUF-RN and M registers (for instance 2048 bit) and an L-bit register for the produced b. M consists of exactly L-bit ones distributed randomly over the bit locations in the M register, as illustrated earlier in Figure 17. The M and PUF-RN registers are shifted/rotated right and the bit shifted from M is checked if it is equal to one or not. If the value is equal to one, the value of the bit shifted from PUF-RN is shifted to the *b* register, otherwise it is discarded. Therefore, after n cycles, the *b* register will be holding the L-bit random number to be used in modular exponentiation for Diffie–Hellman key exchange and SHA3. To illustrate the operation of the masking circuitry, Figure 17 shows an example of producing a 4-bit *b* with n equals 8 and L equals 4. Note that since b is 4-bit, M contains exactly 4 1s.

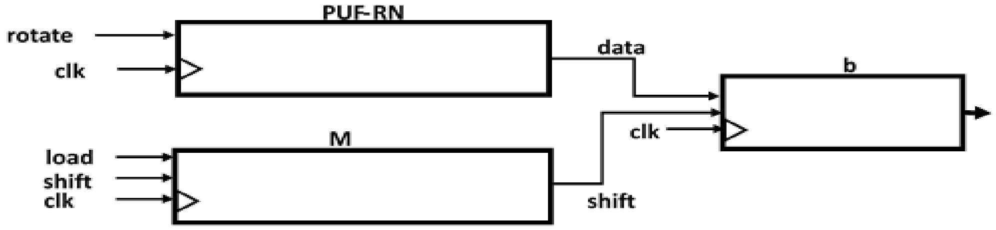


Figure 16: The masking circuitry.

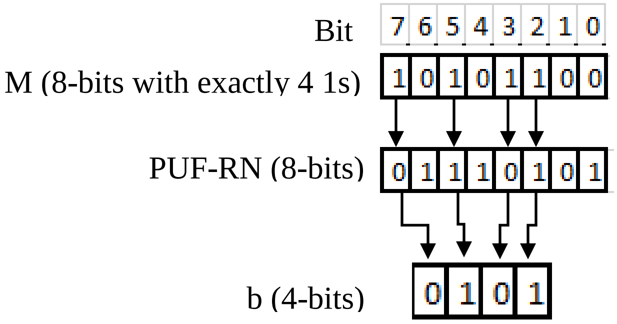


Figure 17: An example of producing an L-bits b from n-bits PUF-RN and M; n in this example equals 8 and L equals 4.

The TA role in this masking is just a simple shuffle of the M upon new M request. Then, TA can send the resulting shuffle, i.e. M, to the client with the hash value of the register b that is resulted when using such produced M in the masking circuitry. The TA also implements its management database for preventing the repetition of Ms and tracking them for each client. A sample implementation of the TA can be found in Appendix D.

4.2.6 Security Analysis

The proposed protocol as illustrated in Figure 14 and Figure 15 provides FPGA authentication, configuration integrity check and data confidentiality in the cloud environment. It provides one-way authentication only; the client authenticates the FPGA while the FPGA does not authenticate the client. The CP is responsible for authenticating the client. It also minimizes communication between the client and the TA.

As was explained above, the steps of the protocol are repeated for every session, the session mask (M) is never repeated for stronger protection and b is never disclosed as a plaintext to the client or to any other party. This enable the use of Ephemeral Diffie-Hellman yielding Perfect Forward Secrecy (PFS). Therefore, even if the FPGA internal PUF-RN is leaked, all previously encrypted exchanges remain secure as the session keys cannot be re-created. Values of a and b are deleted by the Client and the FPGA, respectively, once each side establishes the session key. Also, all session keys are deleted at end of session. Furthermore, the width of the hash function output should be at least twice the size of the generated session key to provide strong collision resistance [143].

Figure 18 illustrates how FPGA impersonation is prevented by providing the hash of the b and the ID(Fi). Let an attacker (t) try to impersonate an FPGA with ID(Fi). The TA sends the client the hash Hash(b(M) + ID(Fi)) which must match the hash received from attacker *t*. In this case, the client receives Hash(b(M)* + ID(Fi)) and g^t mod p which do not equal Hash(b(M) + ID(Fi)) and g^b mod p. Hence, Fi impersonation by the Attacker (t) is prevented, and replaying the hash to be sent by the Fi is also prevented because M is never repeated. Integrity checking is also secured through the use of the symmetric session key, and replaying it is prevented through the use of a newly client-generated RN.

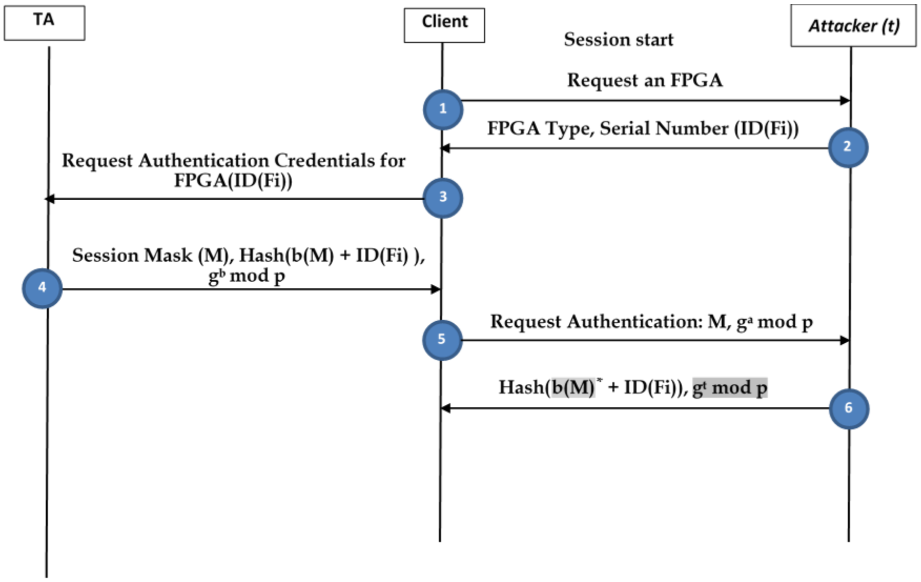

TA
Client
Attacker (t)
Session start
Request an FPGA
1
FPGA Type, Serial Number (ID(Fi))
2
Request Authentication Credentials for FPGA(ID(Fi))
3
Session Mask (M), Hash(b(M) + ID(Fi)), g^b mod p
4
Request Authentication: M, g^a mod p
5
Hash(b(M)* + ID(Fi)), g^t mod p
6

Figure 18: FPGA impersonation prevention.

Figure 19 illustrates how the proposed scheme prevents MiM attacks by using the Ephemeral Diffie-Hellman. Such an attack would fail due to inability of the MiM to re-compute the hash sent by the FPGA while providing the correct g^b mod p. Exchanges between the client and the TA (messages 3 and 4 in Figure 19) can be protected by a standard Secure Socket Layer (SSL) protocol. This prevents a MiM attacker from obtaining the hash and the g^b mod p values sent by the TA to the client (message 4 in Figure 19) which are needed to launch a successful MiM attack.

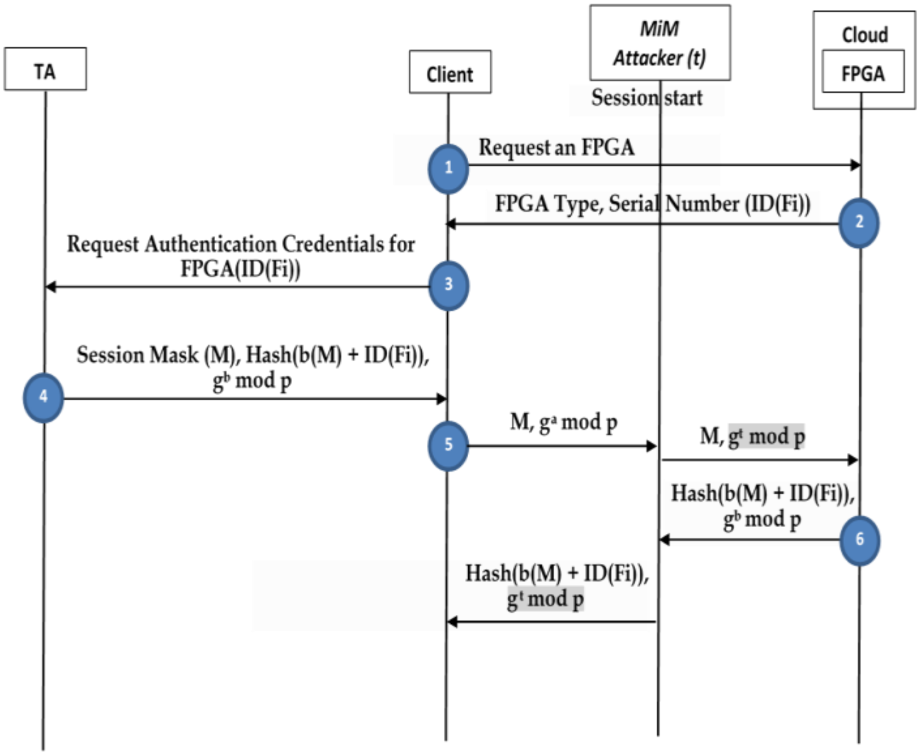

TA
Client
MiM Attacker (t)
Cloud
FPGA
Session start
Request an FPGA
1
FPGA Type, Serial Number (ID(Fi))
2
Request Authentication Credentials for FPGA(ID(Fi))
3
Session Mask (M), Hash(b(M) + ID(Fi)), g^b mod p
4
M, g^a mod p
5
M, g^t mod p
Hash(b(M) + ID(Fi)), g^b mod p
6
Hash(b(M) + ID(Fi)), g^t mod p

Figure 19: Man-in-the-middle attack prevention.

Invasive attacks are useless as the effort to de-package and read out stored keys during operation from one device cannot be used with other devices due to the use of PUFs that generates unique PUF-RNs. The invasive physical attack requires very sophisticated capabilities that are only available to few states and major microelectronic manufacturing companies. The same level of difficulty is true for Fault attack [144].

Non-invasive (including side-channel attacks) and Semi-invasive attacks are also prevented since no key is used more than once and there are not enough data (traces) to perform the analysis. Moreover, no plaintext version of any encrypted data is ever made available to any party other than the one who generated it. To protect most of the physical attacks against the masking circuitry, which is introduced in our scheme, the PUF-RN could be inverted and stored in another register, Figure 20, such that the power traces expose nothing more than a monotonically increasing current to the attacker. That is, every shift to the b and its complement registers produces the same current. Also, the values in these registers are shifted; adding the current of these shifts to the newly shifted bit and making the current monotonically increasing.

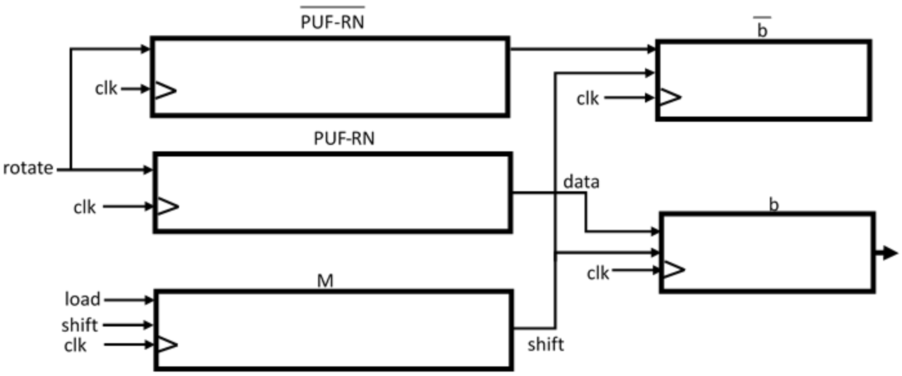


Figure 20: A masking circuitry resistant to physical attacks

For protecting other components of the static logic such as AES from physical attacks, several methods already exist for this purpose such as [145] [146] [147] [148]. Table 4 summarizes the attacks discussed in Section 0 and our countermeasures against them. Most network attacks target denial of services (data confidentiality is guaranteed using our scheme) and they can also be mitigated using cloud protection services [149].

Table 4: Attack types and our countermeasures

Attack category	Attack type	Countermeasure
Cryptographic attacks	known-plaintext	AES is resistant to this attack and the session key is needed by the attacker. Also, plaintext is not exposed to any party.
	chosen-plaintext	AES is resistant to this attack and the session key is needed by the attacker. Also, plaintext is not exposed to any party.
	ciphertext-only	This attack is the hardest to achieve because the attacker has no knowledge of anything except the ciphertext.
	chosen-ciphertext	AES is resistant to this attack and the session key is needed by the attacker. Also, plaintext is not exposed to any party.
	related-key	This attack is prevented since the PUF is random, unpredicted and unclonable
	known-key distinguishing	This attack is not possible as the attacker must be a client to send to the FPGA
Network attacks	wiretapping	Since the data is encrypted, this attack is prevented
	port scan	This attack is not applicable to FPGAs as the attacker gains nothing from scanning the ports
	idle scan	This attack is not applicable to FPGAs as the attacker gains nothing from scanning the TCP ports in case TCP

			protocol is used
	Man-in-the-middle		MiM attack is prevented as shown in Figure 19
	impersonation		FPGA impersonation is prevented as shown in Figure 18
	replay		Replay attacks are prevented since M is never repeated and the use of client-generated RN
	ARP poisoning		ARP poisoning is prevented because the data is encrypted
	ping flood		This attack is not applicable to FPGAs
	ping of death		This attack is not applicable to FPGAs
	Smurf attack		This attack is not applicable to FPGAs
Physical attacks	miroprobing		This attack is expensive as discussed in this section
	side-channel	cache	This attack is not applicable to FPGAs
		timing	This attack is not applicable to FPGAs
		power analysis	This attack is prevented using the masking circuitry in Figure 20
		electromagnetic	This attack is prevented using the masking circuitry in Figure 20
	semi-invasive		semi-invasive are prevented using the masking circuitry in Figure 20
	brute force		This attack requires exponential time and can become infeasible if key length and hash function output are long enough
	fault injection		This attack is expensive as discussed in

			this section
	data remanence		This attack is not applicable to FPGAs
	FPGA	Reverse engineering	Reverse engineering is prevented since readback is not possible
		tampering	The Static Logic is installed by the TA and cannot be readback. The user application is sent to the FPGA in the cloud encrypted and it is decrypted inside the FPGA by the Static Logic after authenticating the FPGA. Furthermore, plaintext configuration is never obtained by any party other than the one that created it. Even if an attacker managed to install HW Trojans (after the clients configure their application), the repeated integrity checks would expose that to the client.
		cloning	Our system prevents such an attack due to the use of PUFs, which produce a unique PUF-RN for every FPGA. Hence, cloning a configuration to another FPGA will result in incorrect b and the FPGA authentication will fail, as illustrated in Figure 19. In addition, the user can securely perform periodic integrity check to ensure that the FPGA is not modified while running the application.
		counterfeiting	same countermeasures of cloning attack can be applied to prevent counterfeiting
		crippling	This attack is prevented in our scheme

			as the attacker must obtain the session key to send the invalid bitstream to the static logic for partial reconfiguration. Further, the static logic cannot be erased because external configuration is disabled.

To verify the protocol and check if there are any vulnerabilities, a formal verification was carried. The tool used for verification is ProVerif, which is a well-known tool to verify security protocols [150]. ProVerif can analyze security protocols automatically under the assumption that the attacker is active; meaning that the attacker can send, receive and modify messages. ProVerif proves the secrecy (the attacker cannot obtain the secret), authentication and strong secrecy (the attacker cannot learn the changes made to the secret). The protocol was written using Pi calculus that is supported by ProVerif. The FPGA generation of the hash and g^b mod p was done by the TA sending them to the FPGA so that the comparison in the client side is correct. The communication between the TA and the client/FPGA was secured using a shared key using DH. The output produced by ProVerif showed that our protocol is secure. The code of our protocol along with detailed comments can be found in appendix B.

4.3 Experimental Results

In this section, the details of the FPGA prototype and the OpenStack cloud infrastructure are provided. The section introduces OpenStack and its components, integrating FPGAs with OpenStack components and the performance evaluation of our prototype.

4.3.1 Background on OpenStack

OpenStack is a group of open source projects aimed to provide comprehensive cloud services. There are six main components of OpenStack summarized in this section including:

1. **OpenStack compute (called Nova):**

   Nova is the component that manages the Infrastructure as a Service (IaaS) cloud computing platform. It includes drivers that interact with the underlying virtualization. Informally, it is the worker that deploy virtual machines in to the hosts by using the virtualization layer (hypervisors). Nova is the most complicated component of OpenStack [28] because it deals with external hypervisors. The components of nova compute are:

– Nova-api: accepts and responds to user's compute API calls

– Nova-conductor: acts as an intermediary between the compute node and the database node. This is to make the communication to the database from the compute nodes more secure.

– Nova-scheduler: takes VM requests from a common queue and determine to which host it should go.

– Nova-compute: creates and terminates VMs through hypervisor APIs such as KVM and Xen.

2. **OpenStack image (called Glance):**

   Registering, discovering and retrieving virtual machines is done by using the glance component. This component is generally responsible for any operation related to the virtual machine image.

3. **OpenStack Networking (called Neutron):**

   OpenStack Neutron is the network component of OpenStack. It is scalable and can be deployed in a separate server to scale the cloud.

4. **OpenStack Dashboard (called Horizon):**

   OpenStack Horizon provides a graphical interface to the users and cloud administrators to access, monitor, and automate cloud resources.

5. **OpenStack identity (called Keystone):**

   OpenStack Keystone is the identity management component of OpenStack and it authenticates the cloud components and cloud users.

6. **OpenStack block storage (called Cinder) and OpenStack object store (called Swift):**

   OpenStack Cinder and Swift provide storage to OpenStack virtual machines. The difference between Cinder and Swift is that Swift is storing metadata related to objects while Cinder stores user data attached to a VM as blocks.

These components are open source python codes and they communicate by using a central database installed in the controller node and use messaging software called rabbit to pass information between them. The relationship between the seven components is complicated as illustrated in Figure 21. All components are connected to identity services and all components are accessible using the dashboard service.

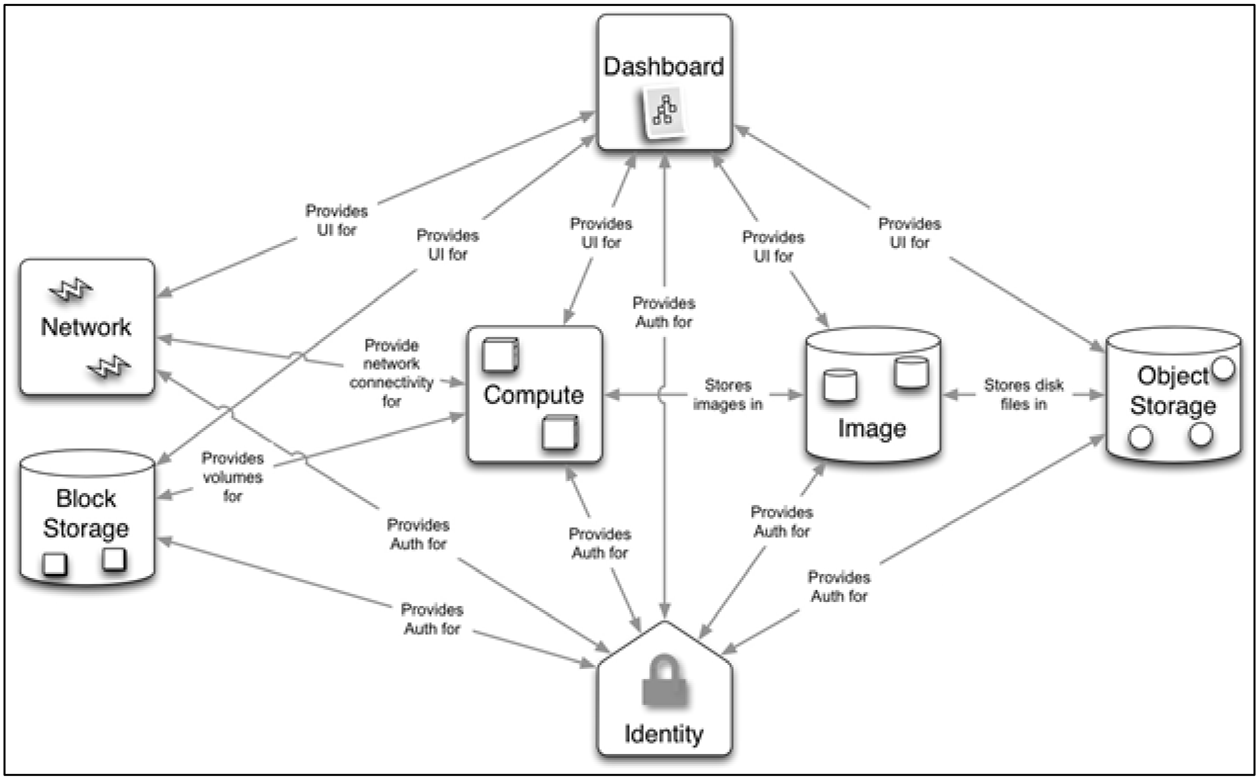


Figure 21: OpenStack Architecture shows the seven main components of OpenStack and communication between them [28]

4.3.2 Testbed Implementation

To evaluate the practicality and performance of the proposed scheme, a complete proof-of-concept prototype of a cloud-based FPGA system has been implemented using OpenStack's Juno release [151]. OpenStack was chosen because it is an open source, allowing it to be modified to integrate FPGAs into cloud infrastructure. It is currently used by about 70 % of cloud operators [152] and is highly ranked by researchers [153]. Figure 23 shows both, the logical cloud infrastructure as per OpenStack guide [28], and the implemented physical testbed. It consists of three nodes; Compute, Network, and Controller nodes. Two Intel PowerEdge servers each with 16 cores, 32 Gb of RAM and 700GB hard disk were used for the compute and the network nodes while an i5 PC with 4Gb of RAM and 500GB hard disk was used as a controller. Two Fixed Configuration Ethernet switches with 16 Gbps forwarding bandwidth, 32 Gbps switching bandwidth, and 64 MB/32MB DRAM/Flash memories provided the interconnection fabric within the cloud. SW1 is used for cloud management and SW2 is used for clients' communications with their VMs and FPGAs. The cloud was attached to a LAN via the Network node as shown in Figure 23. There are two network interfaces in each of the Compute and Network nodes to setup the private networks necessary to setup the cloud. A 3rd interface is in the Network node for the virtual machines (VMs) to communicate externally and only one interface is needed in the Controller node to monitor the VM instances. Optional components are implemented in the compute node. The network and compute nodes contain br-int and br-ext (internal and external bridges, respectively) that are used to share the network interface to enable users to communicate with their virtual machines. No legacy networks were used in this work nor a storage network since storage is not implemented separately. This basic setup can be easily scaled up to thousands of compute nodes and additional networking nodes as needed.

A Xilinx Virtex-6 LX 550T FPGA prototyping board, Figure 22, (with 1 Gbps Ethernet ports) was attached to the cloud as an autonomous HW resource using Python scripts that implement the driver-agent model supported by OpenStack as outlined in details in [154]. The FPGA is then scheduled and assigned to a client as a conventional VM and all the client traffic are forwarded to it.

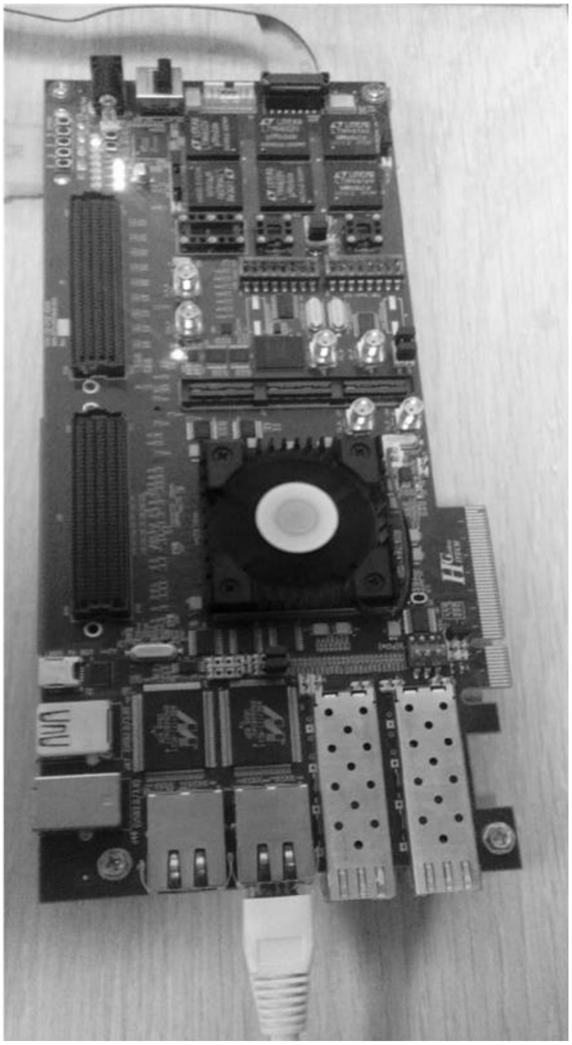


Figure 22: Xilinx Virtex 6 XC6VLX550T board

A special user interface software that would run on a client's workstation was developed. It manages the setup and operation of an FPGA-based computing node on the cloud. It handles all the communications between the client's workstation and the on-cloud FPGA. Clients can use it to establish/manage a session on the cloud-based FPGA and handle all data transfers to/from the FPGA from/to the client's workstation. It also handles all the messages with the TA (using a special port), performs the key generation, encryption and decryption of the FPGA's configuration files, client's data and results. The interface also reports the total time elapsed in establishing the secure FPGA-based computing node on the cloud. The trusted authority’s server was emulated by a Python script running on the TA workstation on the LAN. It receives the client’s request, performs a random shuffle to produce M, compute g^b mod p and the hash Hash(b(M) + ID(Fi)), and send them to the client. Since the TA is emulated within the same LAN as the client, Amazon cloud is ‘pinged’ to estimate the latency of communicating with the TA. The Ping is done for different Amazon cloud locations using the CloudPing.info service [122]. On average, the ping command took around 290 ms from the testbed site. This represents the round-trip time taken to obtain the FPGA mask (M), g^b mod p and the corresponding hash value from the TA.

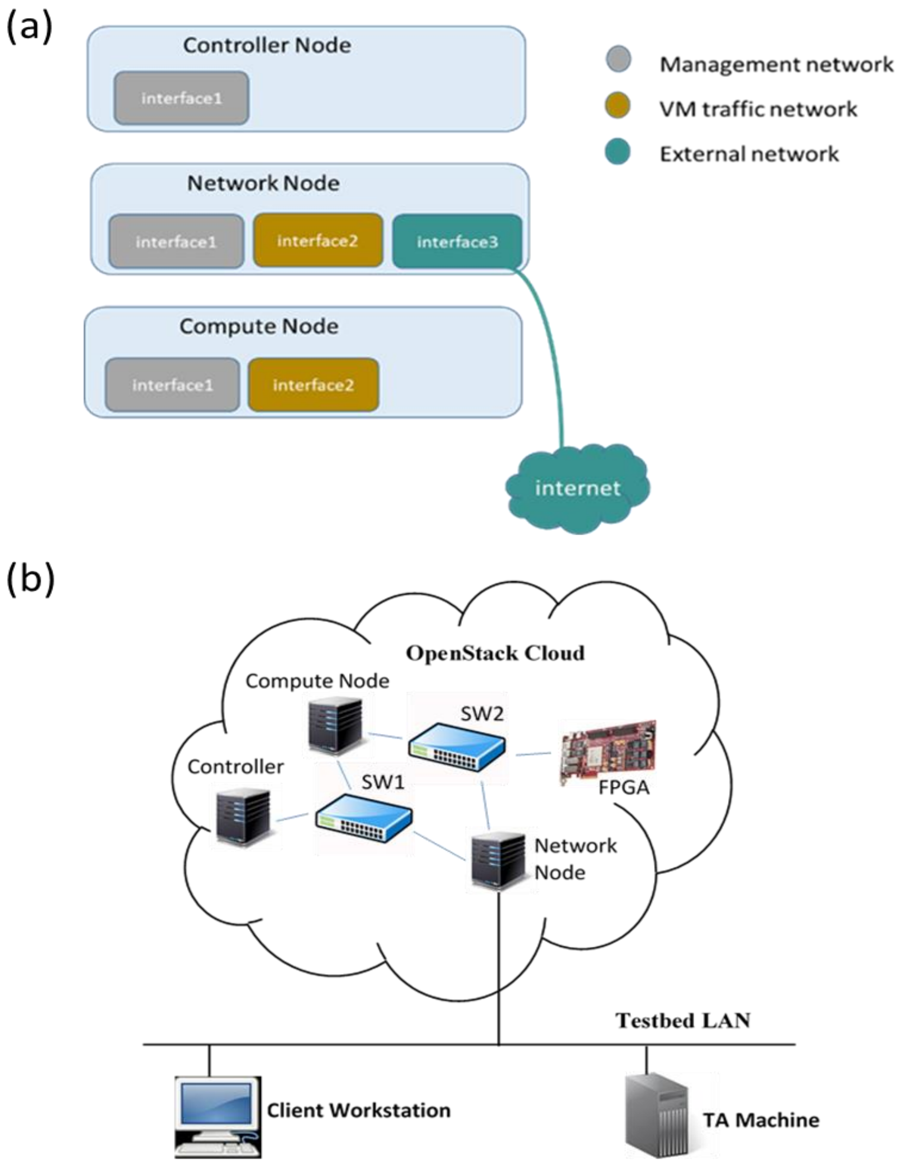


Figure 23: The implemented OpenStack cloud. (a) OpenStack Cloud implementation logical architecture, (b) OpenStack Cloud implementation physical implementation.

For prototyping purposes, the Static Logic blocks were implemented using the FPGA's reconfigurable logic blocks. The Static Logic is made of the following components (a detailed description of the Static Logic components can be found in Appendix A):

1. A 512-bit SHA3 hashing block to support 256-bit session keys. This circuit was designed and implemented based on the Keccak sponge function reported in [155]. The design required major changes to make it routable and to pipeline it (mainly rounds steps),
2. A 256-bit AES crypto-engine based on an OpenCores core by M. Litochevski, and L. Dongjun [156],
3. An OpenCore implementation of the modular exponentiation block (modexp) based on the Square-and-Multiply algorithm by McQueen [157],
4. The PUF as a 2048-bit register containing a random number, and the masking circuitry (as shown in Figure 16),
5. A main FSM to control the various signals and interactions of the static logic. An abstracted drawing of the FSM is shown in Figure 26.
6. An Ethernet controller and a state machine to handle the data flow between the components. Sending and receiving packets FSMs are shown in Figure 24 and Figure 25 respectively.

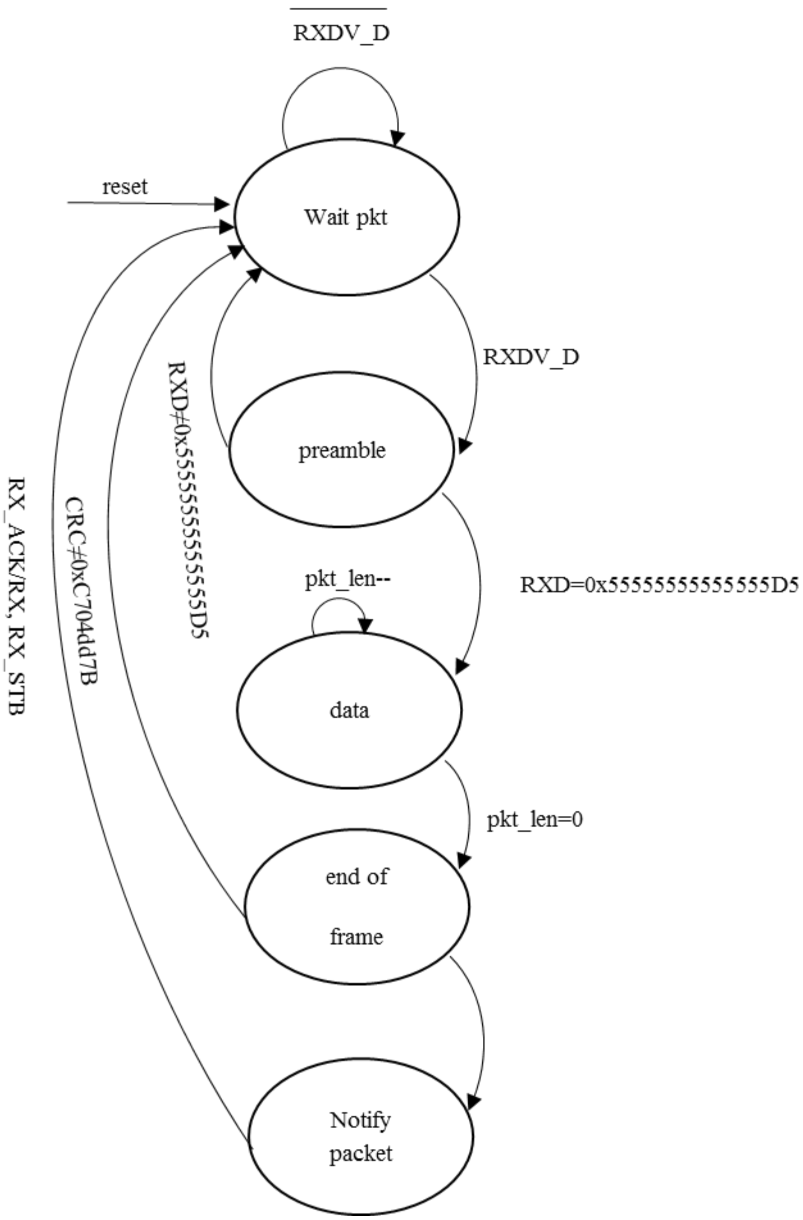


Figure 24: Receiving packets FSM

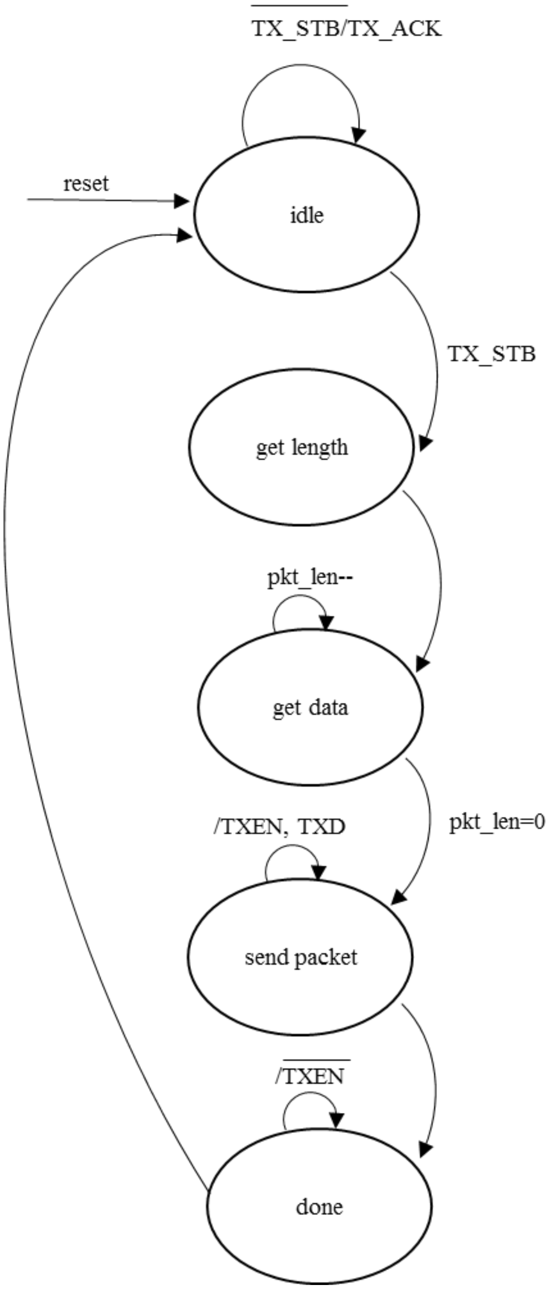


Figure 25: Sending packets FSM

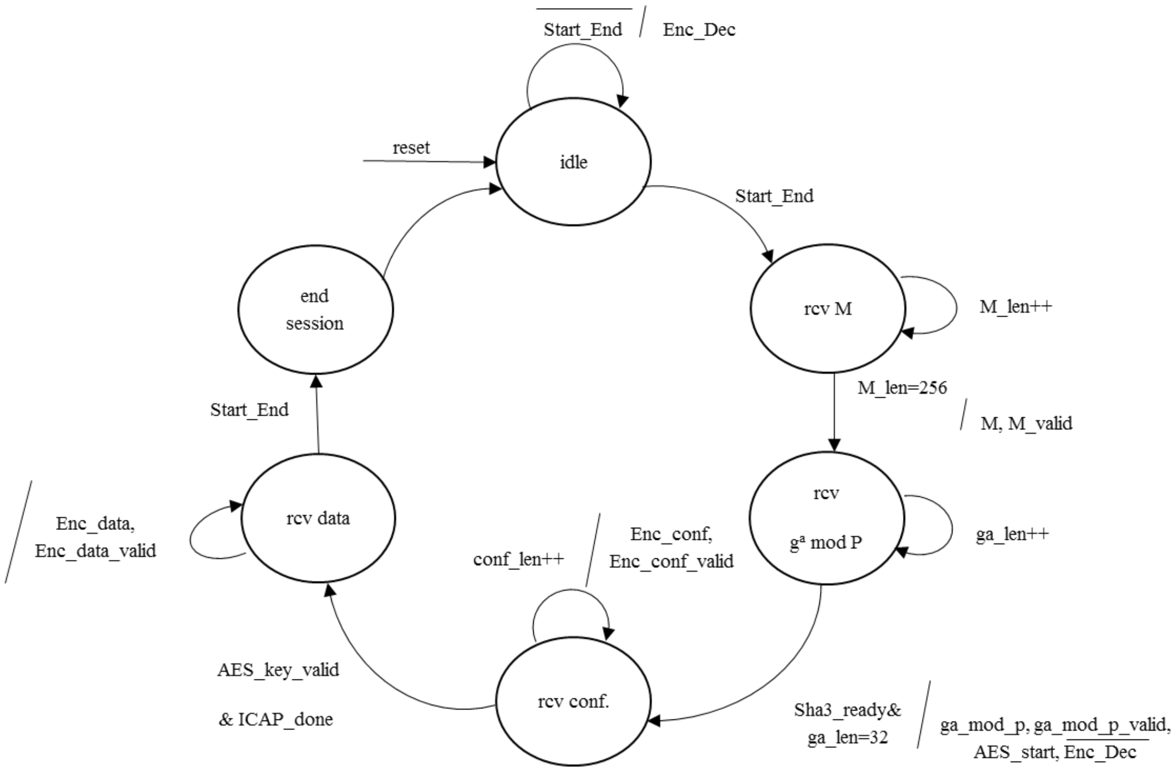


Figure 26: Main FSM

The FPGA’s logic and memory utilization of the different Static Logic blocks are shown in Table 5 along with their maximum possible frequencies. These results show that even if the Static Logic components were to be implemented using the FPGA’s configurable resources they would consume relatively very low resources (~5% of LUTs, ~2.5% of flip-flops, ~1.1% of the available block RAMs, and ~0.5% of the available DSP multipliers). Prior work ([158]–[160]) reported similar results indicating that these types of functions can be implemented very efficiently on FPGAs.

The Static Logic was also synthesized as a custom circuit to estimate its area if it was made as hard macros on the FPGA. The total gate count was 144,012 gates (total RAM and FFs count remain the same as the FPGA implementation). Based on that, and to put this into perspective, the total area of the Static Logic as custom HW macros is estimated to be 0.0414 mm2 in a state-of-the-art 16/14 nm fabrication technology based on the International Technology Roadmap for Semiconductors (ITRS) [161]. A typical state-of-the-art FPGA would have a die area from few hundred mm2 to around 2,000 mm2 [162].

As shown in Table 5, the Static Logic synthesized on the FPGA was also relatively fast. All components used the 100 MHz FPGA board clock since that was more than enough to handle the board's 1 Gbps Ethernet traffic. The SHA3-512 achieved a throughput of 237MB/s and a latency of 27 cycles to process 64B of data. Hence, the extra hashing step for the variant protocol of Figure 15 will only add 27 cycles to the authentication time. Similarly, the AES-256 module had a throughput of 235MB/s and 40 cycles latency for 16B of data. In fact, it only takes 17ms to encrypt/decrypt a 4MB file. Modexp component is rarely used and it is used only at the beginning of the session. It takes less than 0.7ms to perform modular exponentiation for 256 bit base and exponent with the 256 bit modulus. The latency of our basic masking circuitry is 2048 cycles for the 2048 bit PUF-RN. These components can be easily operated at higher frequencies to handle higher bandwidth Ethernet links.

Figure 27 illustrates the required behavior of the on-FPGA Static Logic. Upon receiving M and g^a mod p from the Ethernet controller, M_valid and ga_mod_p_valid signals should go high for one clock cycle. M_valid strobes the masking circuitry to produce *b* which would be used by the modexp and SHA3 modules to start producing g^b mod p and the corresponding hashes. g was set to 2 and p to a 256 bit random number. Once Hash(b(M)+ID(Fi)) is ready, SHA3_out_valid signal should go high for one clock cycle.

Similarly, gb_mod_p_valid should go high for one cycle when g^b mod p becomes valid. The FPGA can then receive the encrypted configuration, decrypts it, and then use the ICAP to configure the client partial region.

Table 5: Static Logic resource consumption.

Static Logic	LUTs	FFs	BRAMs	DSP	F_{Max} (MHz)
Full System	16,546 (4.81%)	17,488 (2.54%)	14* (1.11%)	4 (0.46%)	212.8
SHA3-512	7,573 (2.20%)	2,211 (0.32%)	0 (0.00%)	0 (0.00%)	273.9
Ethernet Controller	1,302 (0.38%)	1,045 (0.15%)	12 (0.95%)	0 (0.00%)	234.6
AES-256	4,068 (1.18%)	1,215 (0.18%)	2 (0.16%)	0 (0.00%)	264.0
modexp	6,816 (1.98%)	3,595 (0.52%)	0 (0.00%)	4 (0.46%)	130.6
Masking circuitry	3,100 (0.90%)	4,349 (0.68%)	0 (0.00%)	0 (0.00%)	430.3
FSM	2,488 (0.72%)	2,460 (0.36%)	0 (0.00%)	0 (0.00%)	413.6

* ~ 264 Kb out of 22,752 Kb total.

Figure 28 shows the actual signals obtained from the implemented prototype using Xilinx Chipscope (a technology that allows real-time monitoring of on-FPGA buses). Only the least 32 significant bits of each bus are displayed in Chipscope since the maximum triggers that can be shown in Chipscope is 256 bits. In addition, the signals were captured from three successive runs. In the 1st run (Figure 28(a), M is first received, then g^a mod p is received, *b* is generated by the masking circuitry, then hash(b+ ID(Fi)) is computed and sent back to the client. For the 2nd run, Figure 28(b), both M and *b* are set as constants to the values that were sent/computed in the 1st run, then g^b mod p is computed followed by g^{ab} mod p. However, since each modular exponentiation operation takes about 70,000 cycles, ChipScope is only triggered before the last 100 cycles or so g^b mod p generation (g^{ab} mod p generation is not shown). Finally, for the 3rd run, Figure 28(c) shows receiving the encrypted configuration and decrypting it (the Chipscope output cursor position shows the decrypted configuration synch word “665599AA”). As this Figure shows, the implemented Static Logic achieves the correct required behavior.

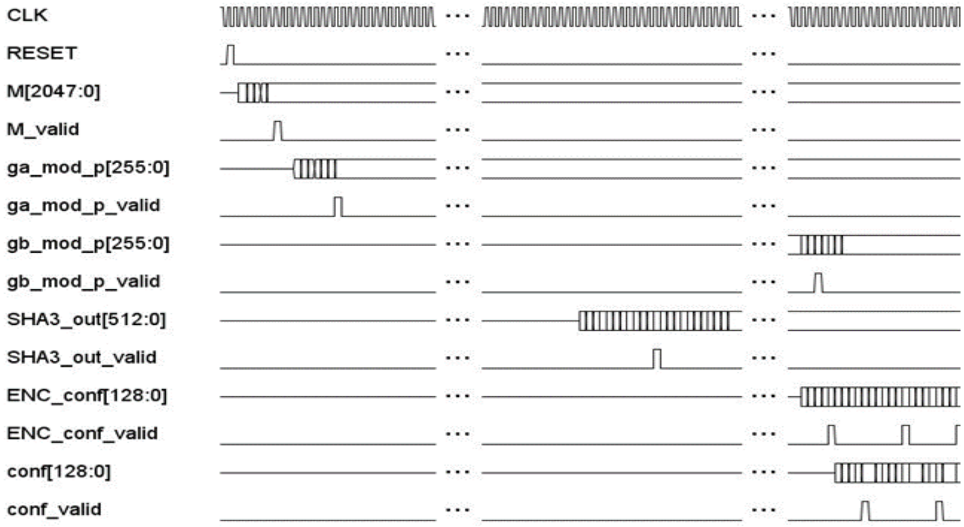


Figure 27: The expected behavior of the Static Logic.

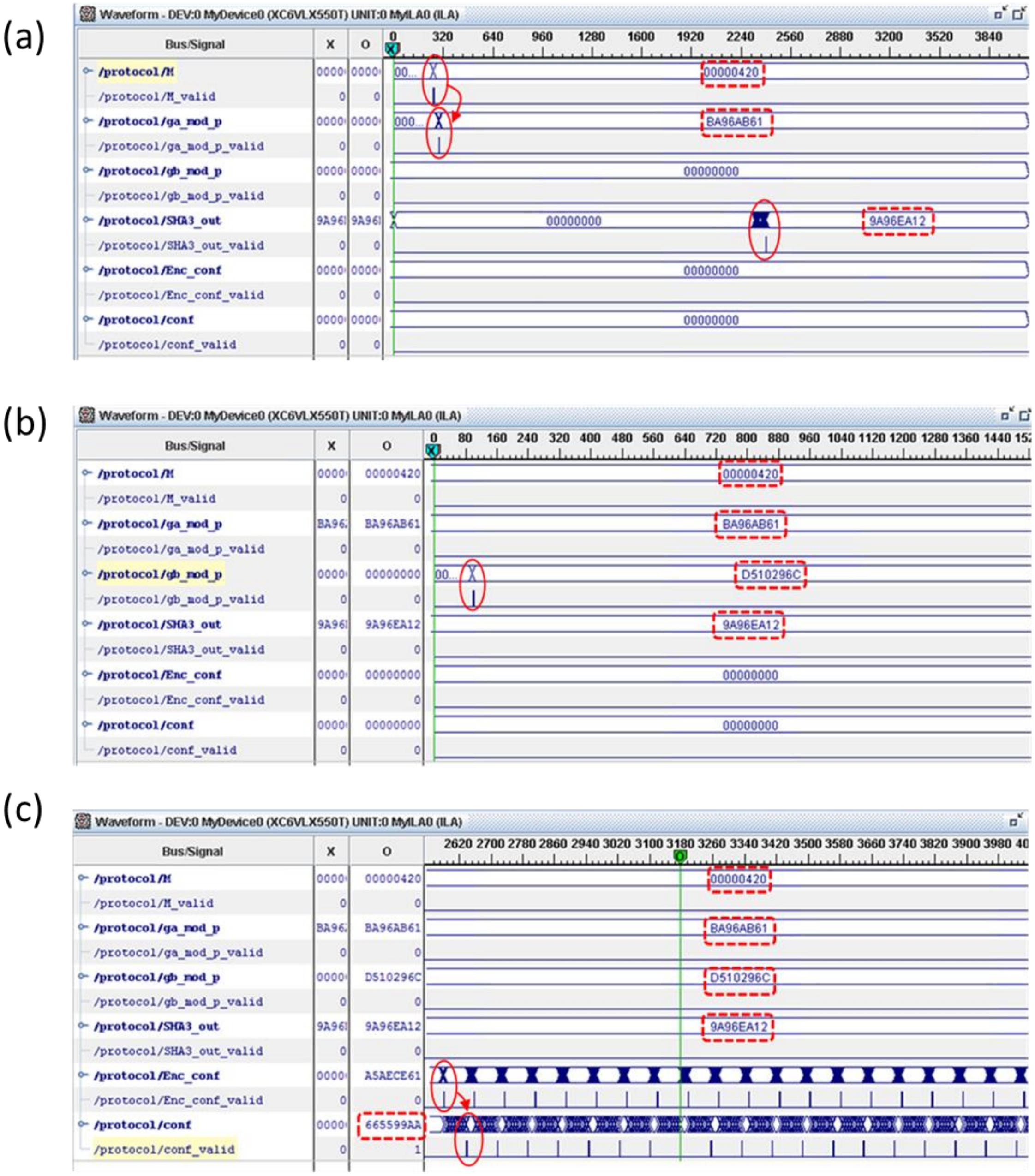
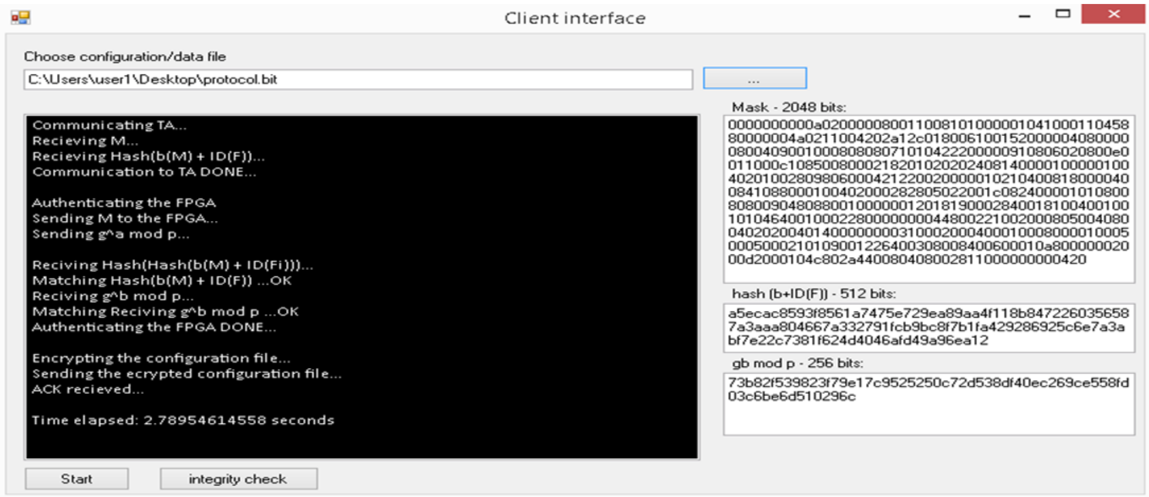


Figure 28: Chipscope screenshots showing the various signals of the implemented Static Logic. (a) Receiving M, ga mod p and producing the hash value, (b) gb mod p generation, (c) Receiving the encrypted partial configuration. The output cursor points to the beginning of the configuration (665599AA).

To evaluate the practicality of the proposed scheme, the setup time of an FPGA-based computing node on the cloud was measured and compared to conventional SW-based virtual-machines image boot time on the same cloud. Figure 29 shows a snapshot of the user interface SW. It shows the sequence of events to establish a secure session on the FPGA. Table 6 shows the boot times for various virtual machine instances with CirrOS images on the OpenStack-based cloud implementation testbed. The boot requests were issued from a client workstation on the same LAN as the cloud testbed. CirrOS is a 12 MB OpenStack small Linux based operating system image that is used for testing images in OpenStack clouds.

As Table 6 shows, it took 41 seconds to boot a medium size VM within the same cloud. Using the same client-cloud configuration, a secure FPGA-based computing node (with 10 MB configuration file) is booted in about 2.8 seconds as shown in Figure 29. This is about 15 times faster than a medium size VM on the same cloud. Moreover, considering a client having an internet connection with a speed equals to the global average speed of the internet (i.e. 6.3Mb/s [163]) he/she can establish a session from his/her location to the on-cloud FPGA in about 14 seconds.



Choose configuration/data file

C:\Users\user1\Desktop\protocol.bit

```
Communicating TA...
Recieving M...
Recieving Hash(b(M) + ID(F))...
Communication to TA DONE...

Authenticating the FPGA
Sending M to the FPGA...
Sending g^a mod p...

Reciving Hash(Hash(b(M) + ID(Fi)))...
Matching Hash(b(M) + ID(F)) ...OK
Reciving g^b mod p...
Matching Reciving g^b mod p ...OK
Authenticating the FPGA DONE...

Encrypting the configuration file...
Sending the ecrypted configuration file...
ACK recieved...

Time elapsed: 2.78954614558 seconds
```

Mask - 2048 bits:

```
0000000000a020000080011008101000001041000110458
80000004a0211004202a12c018006100152000004080000
080040900100080808071010422200000910806020800e0
011000c1085008000218201020202408140000100000100
40201002809806000421220020000010210400818000040
0841088000100402000282805022001c082400001010800
80800904808800100000012018190002840018100400100
101046400100022800000000044800221002000805004080
04020200401400000000310002000400010008000010005
00050002101090012264003080084006000010a800000020
00d2000104c802a4400804080028110000000000420
```

hash (b+ID(F)) - 512 bits:

```
a5ecac8593f8561a7475e729ea89aa4f118b847226035658
7a3aaa804667a332791fcb9bc8f7b1fa429286925c6e7a3a
bf7e22c7381f624d4046afd49a96ea12
```

gb mod p - 256 bits:

```
73b82f539823f79e17c9525250c72d538df40ec269ce558fd
03c6be6d510296c
```

Figure 29: The user C# interface showing the message exchanged during session establishment.

Table 6: Boot time (in seconds) for different virtual machine sizes on the implemented OpenStack cloud.

VM Size	Virtual cores	RAM	Disk	Ephemeral Storage	Boot time(sec)
xlarge	8	16 GB	10 GB	160 GB	52
large	4	8 GB	10 GB	80 GB	46
medium	2	4 GB	10 GB	40 GB	41
small	1	2 GB	10 GB	20 GB	34
tiny	1	512 MB	1 GB	0 GB	30

To evaluate the performance of our scheme and compare its time with Intel SGX, an image processor was used as a user application. The image processor was compiled from c using chips2 [164] and is performing images (bitmap format) edge detection using Sobel operator [165]. The processor was installed and connected with a python script to send the images and receive the output. The design work at the Ethernet speed of at least 80 Mbytes/s. The synthesis report of the image processor is in Table 7.

Table 7: The image processor resource utilization

FF	560 (0.0815 %)
LUTs	1278 (0.3719 %)
BRAMs	11 (1.7405 %)
Frequency	199.233MHz

The experiments for measuring SW (python) performance were carried on a Xeon machine with the following specifications; Intel Xeon CPU with 8 core 3.20GHz, 23.5 GB of memory, 2 TB of disk, and 64-bit Ubuntu 14.04 OS. The experiments were carried over 1G bits of data and Table 8 reports the time it takes for different percentage of the 1G bits of data, where normal python script time to process the data in plaintext is 6.03 seconds. FPGA_1GE is an FPGA with 1 Gbps Ethernet and FPGA_10GE is an FPGA with 10 Gbps Ethernet. Compared to Intel SGX results obtained from [166], our solution is much faster for larger data making it more suitable in terms of performance to cloud applications and streaming analytics. Results of [166] depicted that a data larger than 8 MB (which is the L3 cache size) will make the SGX 5.5x slower due to the overhead of cryptographic operations performed while the data leaves the CPU package and a data beyond 92 MB (which is close to the maximum size of 128 MB of the EPC of Intel SGX) will cause Intel SGX to be 200x slower due to the overhead associated with Intel SGX page swapping. An alternative method for processing data larger than 92 MB using Intel

SGX is to keep the data encrypted out of the SXG enclave and the client application enters the enclave when needed. This includes calling the enclave, reading the encrypted data, decrypting it inside the enclave, processing the data, encrypting the results, exiting the enclave and writing the enclave data back to the disk. This way, the data may be processed in smaller chunks; reducing the overhead associated with larger data and avoiding page swapping. However, entering the enclave to process sensitive data is highly dependent on the application using the enclave. Also, entering and leaving the enclave is costly as illustrated by Zhao et al. [127] and considering an enclave call per 8000 instructions leads to an overhead of 467% compared to executing instructions without calling an enclave. In our comparison, we assume the overhead of processing chunks greater than 8 MB and less than or equal to 90 MB to be 5.5x as this is the minimum slowdown reported in [166]. This slowdown is also reasonable for streaming applications (like our image processor) that has no data dependency, which makes it possible to divide the data into chunks less than 92 MB.

Table 8: Performance comparison with Intel SGX

		Time (sec)		
Data size (MB)	% from 1Gb	SGX	FPGA_1GE	FPGA_10GE
134.22	100	33.18	5.03	3.52
120.80	90	29.86	4.53	3.17
107.37	80	26.54	4.03	2.82
93.95	70	23.23	3.52	2.47
80.53	60	19.91	3.02	2.11
67.11	50	16.59	2.52	1.76
53.69	40	13.27	2.01	1.41
40.27	30	9.95	1.51	1.06
26.84	20	6.64	1.01	0.70
13.42	10	3.32	0.50	0.35
12.08	9	2.99	0.45	0.32
10.74	8	2.65	0.40	0.28
9.40	7	2.32	0.35	0.25
8.05	6	1.99	0.30	0.21
6.71	5	0.06	0.25	0.18
5.37	4	0.05	0.20	0.14
4.03	3	0.04	0.15	0.11
2.68	2	0.02	0.10	0.07
1.34	1	0.01	0.05	0.04

4.4 Client's Circuit Intellectual Properties on the Cloud

The problem of protecting Client's Circuit IPs in the cloud is unique and different than protecting the IP in an FPGA owned by a client. The client is the party that sends the encrypted configuration to the cloud (including his application and the IP). Normally, protecting an IP in an FPGA owned by a client (i.e. one client) involves ensuring that only the FPGA that is supposed to use the IP is configured (i.e. prevent cloning the IP), which is not the case for the on-cloud FPGA as the FPGA assigned to the client is unknown to the IP core vendor (CV). Our scheme discussed in previous sections can be used to protect the IP core in the cloud from any party, including the client and the cloud provider. The scheme needs to be modified, as in Figure 30, to be used for protecting third party IPs. Basically, a static logic is needed to receive the IPs and it contains a controller that controls the masking circuitry and the ICAP interface for partial reconfiguration and a decryption module to decrypt the IP and its key. The steps needed to securely partially configure the on-cloud FPGA are as follows:

1. CV enrolls IP to the TA by providing the key IP_key.
2. TA encrypts the key using b that is produced by applying a mask m.
3. TA sends the user M, the encrypted IP_key (Enc(IP_key))and hash of b.
4. CV sends the IP encrypted using IP_key to the client.
5. The user sends M to FPGA i.
6. FPGA i generates b.
7. User sends Enc(IP_key).
8. FPGA decrypts Enc(IP_key).

9. Users sends encrypted IP.

10. FPGA decrypts it and uses ICAP to configure the partial region of the IP.

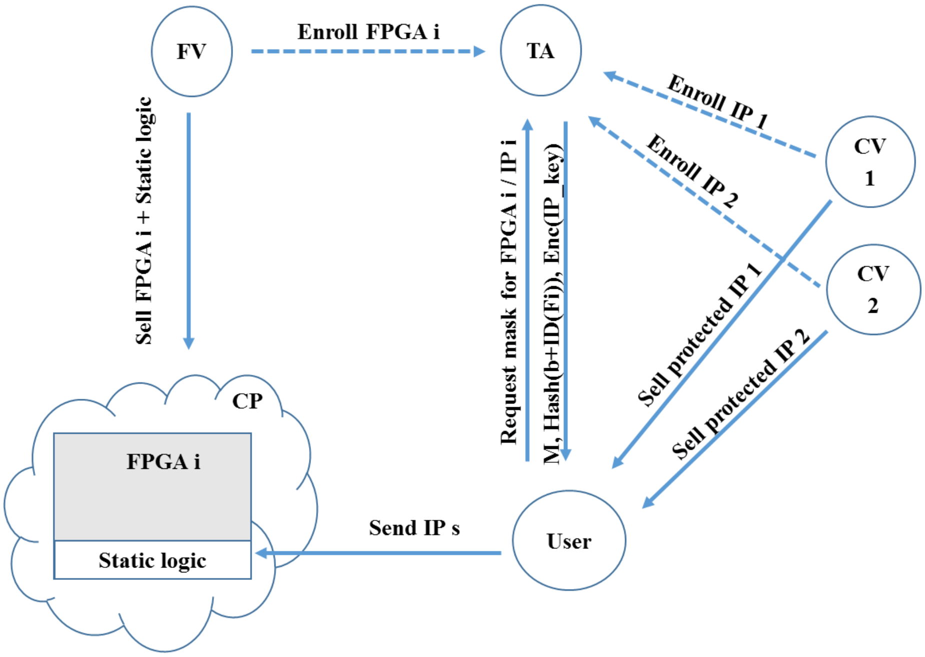


Figure 30: The framework for protecting IPs in the cloud

The CV might need to check for integrity of its IP while running in the FPGA and a new mechanism should be developed to provide such checking. In our scheme, the hash of the IP should be sent to the client encrypted using the CV private key so that only the CV can decrypt the hash.

The infrastructure for this modified scheme is similar to the original scheme when used for protecting client data who use IPs in his design from third parties, and only the main state control should be modified to accept more than one M and to store the CV's private key. It is observed that this modified scheme is extending our proposed original scheme to the visualized FPGAs where multiple clients' configuration can securely run in one

FPGA in the cloud. The FPGA static logic should be the same and only the FPGA management would be changed, as discussed by Stuart Byma and steffan [154].

4.5 Conclusions

In this work, a new FPGA-based scheme for securing users' data (and applications) in clouds is proposed. It was shown that the proposed protocol for establishing a secure session on a cloud's FPGA provides strong protection against various types of attack. A complete proof-of-concept prototype implementation of the scheme showed that it is feasible even with existing FPGAs, simple to implement, efficient in terms of resource utilization and takes less time to boot as compared to conventional software-based virtual machines. The proposed scheme achieves perfect forward secrecy, provides authentication of the on-cloud FPGAs by the clients and integrity checking of client configuration to prevent any modification and/or other FPGA related attacks such as reverse engineering and cloning.

CHAPTER

SECURE DATA PROCESSING FOR CLOUD-INTEGRATED INTERNET OF THINGS USING FPGAS

In this chapter, we describe our novel scheme to secure IoT data processing in the cloud from various kinds of attacks, including attacks from insiders. This includes the protocols, and all the HW and SW components required to implement the proposed scheme. The scheme achieves perfect forward secrecy, provides FPGA authentication, a secure way to establish a symmetric session key between the on-cloud FPGA, the IoT device and the client, and user's configuration integrity check while running in the cloud FPGA. Furthermore, a symmetric proxy re-encryption (PRE) is proposed to suite the publish/subscribe systems of IoT. Not only does the implementation show the feasibility of the proposed scheme (in terms of applicability to current FPGAs), but it shows that it has very efficient resource utilization. Furthermore, synthesis results showed that this infrastructure logic would take a total area of about 0.0380 mm2 in a state-of-the-art 16/14 nm fabrication technology if implemented as custom circuits on the FPGA (i.e. Hard Macros). Experiments also showed that our proposed PRE is best suited in FPGAs for better performance. Our PRE takes less than 6 seconds to transform a ciphertext of size 1 Gb in SW and about 1 second in FPGAs using 1 Gbps Ethernet.

5.1 Introduction

Internet of things (IoT) is penetrating to all physical fields, including homes, manufacturers and urban spaces, and is expected to dramatically grow in near future. According to Gartner report [167], IoT devices are expected to reach around 21 billion by 2020. This massive number generates a massive amount of data that need to be stored, aggregated and processed to make a value of the data produced by the IoT devices. Securing sensitive data collected from IoT devices is a crucial issue that needs to be considered and is one of the top issues in IoT ecosystems [168].

The amount of data collected from IoT devices is very large and the cloud is the natural paradigm for storing and processing such huge data. In fact, leading cloud companies already developed platforms for IoT such as Amazon AWS IoT [169], Microsoft Azure IoT Suite [170], and IBM Watson Internet of Things [171]. To this end, cloud computing can be thought as a marketplace for many services that share and handle the data. When it comes to the security of the data, it is the responsibility of the marketplace owner to provide adequate infrastructure for securing the data and it is the responsibility of the data owner to maintain the security of his data and use the right infrastructure in the cloud.

In addition, IoT follow the publish/subscribe fashion and data come from multiple sources (publishers) and can be processed by any cloud component (subscriber). To secure such data in the cloud, a symmetric proxy re-encryption is needed to convert publisher’s ciphertext to a ciphertext that can be decrypted by the subscriber(s). The proxy re-encryption is needed to avoid decrypting the ciphertext while converting it and the proxy re-encryption should be symmetric to allow using symmetric encryption which is more efficient as compared to asymmetric encryption.

However, the cloud is not fully secure for sensitive data of IoT devices, especially from insider attacks. Sensitive data requires much security mechanisms than any other security critical systems requires such as banks systems. Factories might be damaged and people might be injured or even die when such sensitive data is compromised. In the cloud, the IoT data becomes more valuable and more exposed to attacks when aggregated and processed to be presented.

Compared to conventional software-based systems, FPGA configuration does not require the involvement of operating systems, drivers or compilers, making them suitable to build security solution under more robust attack models and stronger security guarantees. FPGA holds potential to deliver more sophisticated solutions for modern machine-to-machine communication and big data applications [5]. FPGAs can be used to process data in the edge (near the IoT devices) or can be integrated with other cloud HW resources to form flexible, scalable, independent and secure compute resources within the cloud infrastructure. Therefore, clients can securely use FPGAs to perform the computation of their sensitive IoT data in the cloud in a secure manner while utilizing the benefits of the cloud and the fast and secure computation of the FPGAs.

In this work, a novel scheme that utilizes FPGAs to secure IoT data processing and secure the applications that use them in the cloud is proposed. The proposed scheme protects against various types of attacks, provides FPGA authentication, ensures fixed logic and user’s application integrity, and data confidentiality. Architectures for integrating FPGAs into the cloud to implement the proposed scheme have also been developed. Furthermore, a symmetric proxy re-encryption has been developed that suits processing IoT data in the cloud. We also provide a complete prototype of our scheme and the proxy re-encryption in FPGAs and discuss their performance in detail. Our solution is scalable (paramount to any cloud-based computing), thus have great potential for secure cloud-based computing as discussed also by other researchers [139].

In Section 5.2 various business IoT architectures are reviewed and in Section 5.3 an overview of the proposed scheme is introduced, including the protocol, its security analysis, the related framework (HW and SW components), and the proposed proxy re-encryption. Experimental results are presented in Section 5.4. This includes the complete implementation of the proposed scheme and proxy re-encryption in FPGAs and the details of all components, their implementation details and performance figures. It also provides performance comparisons between our FPGA-based proxy re-encryption and a software version of it implemented in python. Finally, conclusions are presented in Section 5.5.

5.2 Cloud-Integrated IoT Security Models

In this section, we discuss business cloud-integrated IoT platforms including Microsoft's, IBM's, Google's, Amazon's and Intel's IoT platforms. The business models are studied because there are no clear research directions when it comes to IoT in the cloud.

Figure 31 shows the architecture of Microsoft Azure cloud for IoT [172]. There are three parties in this architecture, the IoT client (the IoT device), the cloud and the client (personal mobile devices and business systems). The client is connected to the cloud and gives actions (commands) to the IoT device through the cloud (there is no direct channel between the client and the IoT device).

The Components of the cloud are:

- Cloud gateway: A cloud gateway is a system that enables remote communication from and to the IoT devices.
- Provisioning API: to make the device known to the cloud. It includes registering and removing the device from the cloud, activating and deactivating the device.
- Stream processors: typically moving or routing data without any transformation.
- Device state store: stores IoT device information such as its ID and registry record.
- App backend: The application back end implements the required business logic of the solution.
- Solution UX: The solution user experience typically includes a website, but can also include web services and APIs with a graphical user interface in the form of a mobile or desktop app.
- Business integration connectors and gateway(s): The business integration is responsible for the integration of the IoT environment into downstream business systems. Typical examples include service billing, customer support.
- In addition to these components, Microsoft developed the data factory component [3] that is used for data transformation and distribution (e.g. MapReduce).

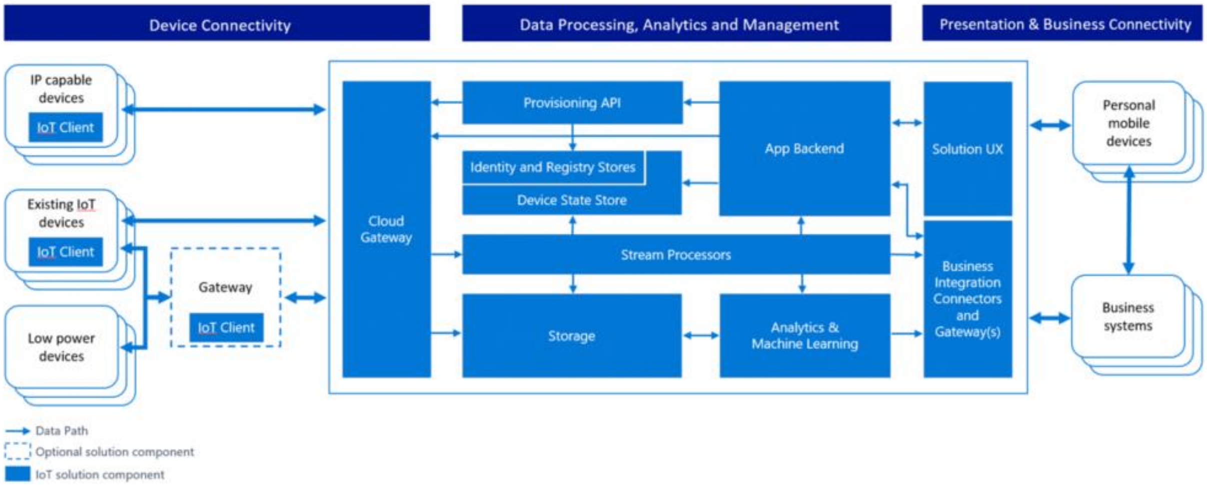


Figure 31: Microsoft's Internet of Things security architecture [172]

The IBM [173], Google [174], Intel [175] and Amazon [169] IoT architectures are similar to the Microsoft’s architecture and they use what they call pipelines to collect and aggregate data from the IoT devices. It can be seen from these architectures that the flow is similar and only the way of handling is different from provider to provider. Further, the data collected from the IoT devices are pre-processed and converted to be processed by an application backend.

Research on IoT security falls mainly into efficiently authenticating IoT devices and securing the end-to-end communication. Due to the impracticality of standard security solutions for authenticating the constrained IoT devices, lightweight authentication methods were proposed. These methods include homomorphism [176], Elliptic Curve Cryptography (ECC) [177] and DTLS protocol based authentication [178]. Commercial cloud based IoT platforms use industry-standard protocols such as TLS and X.509 to secure communication between the IoT devices and the cloud [172]. In addition, several lightweight communication protocols were proposed; including protocols based on public key infrastructure [179], IPv6/6LoWPAN [180], integrating DTLS and CoAP [181] and SSL [182]. Researchers have also proposed various IoT security architectures and discussed technologies that can be used to support IoT security. Layered security architectures were proposed [183] [184] for IoT security and a security verification. The layers cover various techniques related to IoT security such as key management, encryption oracles and access control. These architectures conceptually cover various attacks and mitigation techniques in each layer. A middleware was proposed in [185] to meet the scalability and the high number of heterogeneous devices of the IoT system. The middleware mainly targeted developing a security algorithm to tackle packet sniffing, man-in-the-middle attack and identity spoofing in the IoT environment. An architecture based on lightweight identity based cryptography (LIBC) with elliptic curve cryptography (ECC) was proposed in [186] to solve security issues related to cloud-integrated internet of things environment. However, these architectures neither considered attacks from inside the cloud nor provided any or complete implementations to demonstrate their practicality.

5.3 The Proposed Scheme

In this section, we discuss our scheme that is compliant to the architectures discussed in earlier section. Particularly, we target the IoT data handling with the security added and we also consider the data transformation in secure manner.

In this work, we assume that the on-cloud FPGA devices have the following capabilities, similar to the capabilities assumed in Chapter 4, most of which already exist in current FPGAs:

- Uniquely identifiable using any public identifier such as a printed serial number or other means such as a unique device DNA, similar to that found in Xilinx FPGAs [140]. This nonvolatile, unchangeable and permanently programmed value can be used to authenticate FPGAs running client's configuration. However, device DNA alone is not suitable for device authentication as was illustrated in [141].
- External reconfiguration and readback ports are disabled [142]. External ports such as JTAG and SelectMAP are used to program FPGAs and to read back the configuration in its current state inside an FPGA for debugging purposes.
- Configurable through an internal configuration access port (ICAP) such as in Xilinx devices [142]. An ICAP receives the configuration bit stream from the Static Logic and partially configures another portion of the FPGA. Hence, the FPGA should also support partial reconfiguration,
- Supports readback of static configuration contents such as Look-Up-Tables, interconnects, and I/Os only, but cannot readback dynamic data such as RAM or Flip-Flop contents.
- Have standard high-speed communication ports such as 100 Gigabit Ethernet to enable their in-cloud usage.

We use IoT device to refer to IoT capable device, constrained IoT device or IoT gateway. In case the IoT device is constrained, the IoT gateway is responsible for communicating with the cloud and the FPGAs. Client application and client are used interchangeably in this work.

Figure 32 shows the parties involved in our proposed scheme. In addition to the client, the IoT device and the Cloud Provider (CP) who is providing FPGA-based processing as a service, the FPGA Vendor (FV) who sells FPGA devices to cloud operators also acts as a trusted authority (TA). It is not necessary that the FPGA manufacturer is the TA. Alternatively, an OEM (board) manufacturer could act as the TA. In this case, it will get the PUF-RN from the manufacturer. The cloud is positioned between the clients' applications and the IoT devices and is used for data store and processing and also for command forwarding, as we consider that there is no direct communication between the client application and the IoT device (similar to the cloud business models of Microsoft [172] and IBM [171] for IoT). Therefore, IoT devices receive commands from the cloud and send data to the cloud to be stored and processed and can be viewed by the clients' applications. The processing is secured in the cloud using the FPGAs. Data transformation is handled by the proxy. The trusted authority is an important party that is used to facilitate authentication and secret sharing among the communicating parties.

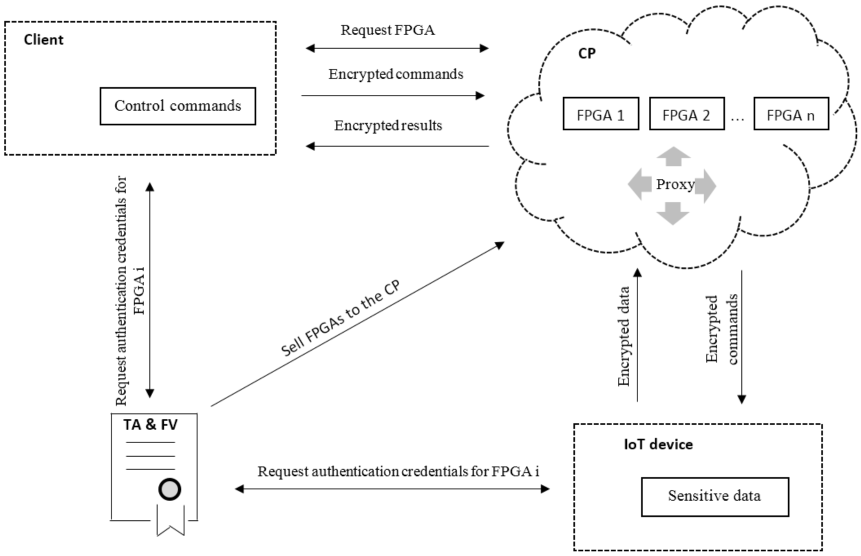


Figure 32: The framework of the proposed scheme.

5.3.1 Description of the Proposed Symmetric Proxy Re-encryption

In order to handle the data from IoT device in the cloud, the cloud takes several steps to make a value of the data that usually comes from different IoT devices and get collected, stored, aggregated, and processed to form the final result. When the data is encrypted, the cloud processing backend would have to use different keys for each data they process and similarly the IoT device needs to send the data encrypted to a specific processing component which is not suitable for the cloud computing paradigm. Therefore, we propose a proxy re-encryption (PRE) that would be in the cloud and would transfer the data encrypted by the IoT devices keys to a data encrypted by the processing components keys and would make it possible to transform IoT data to be processed by any processing component without the need to resend the data by the IoT device.

The proxy re-encryption is shown in Figure 33. User A wants to authorize user B to decrypt the data that is stored or going to be stored in the cloud in the format data* g^r

mod p, where g^r mod p is user's A private key. User A and B first share a session key g^{ab} mod p using Diffie-Hellman (DH) key exchange [26]. User A then sends the re-encryption key (rK) to the proxy that is residing in the cloud. The rK is computed by multiplying the session key by the multiplicative modular inverse of the IoT device private key (g^r mod p). The proxy uses the rK to convert the data by multiplying the data by rK and sends the result to user B. The data is now converted to the format data* g^{ab} mod p and user B can decrypt the data by dividing it by the shared session key (g^{ab} mod p). Observe that the scheme allows any number of users to share the data produced by user A. The division is done using the multiplicative modular inverse and all operation is done with the mod taken and hence the data going in/out to/from the proxy is of the same size. The proof of security of our scheme is straight forward and follows directly from the proof of BBS proxy re-encryption [59].

The PRE can be used in the cloud for IoT data processing assuming that user A is an IoT device and user B is an on-cloud FPGA(s). Therefore, the proxy and the FPGA are in the cloud and they are geographically close to each other, which makes the conversion fast. The only operations that need communication outside the cloud are the FPGA authentication and the sharing of a session key with it. It is worthwhile to mention that using shared key in the PRE is a valid property in our case of using the PRE for IoT data conversion in the cloud since sharing the key should be performed between the IoT device and the on-cloud FPGA.

The properties of the PRE discussed in Section 2.5 that are important in the cloud with FPGAs are highlighted below:

1- Unidirectional: this property is not required in our case.

2- Non-transitive: this is achieved in our PRE as the re-encryption key must be provided by the delegator.

3- Collusion-resistant: this is not important property in the cloud environment and with the use of FPGAs.

4- Non-interactive: The private key is needed to establish a session with the IoT device and our PRE is symmetric.

5- Multi-use/Single-use: this property is not needed in the cloud.

6- Key-privacy: our proxy is part of the pipeline infrastructure in the cloud and this property is not needed as the identity of the IoT device and the FPGA is handled by other cloud components rather than the proxy.

7- Transparent: our PRE is transparent.

8- Key-optimal: our PRE is key-optimal.

9- Temporary: our PRE has temporary property.

Our symmetric proxy re-encryption scheme brings the following advantages when processing IoT data in the cloud:

- The whole process depends on key sharing and not on the data management,
- The scheme is suitable for both the IoT ecosystems and the multi-tenant nature of the cloud computing. Any authorized party can use the data without the data source involvement,
- The FPGAs and IoT devices remain as a worker or as a resource and they are not directly involved in data sharing with applications.

– The PRE makes it possible to secure the publish/subscribe system of the IoT devices and the on-cloud FPGAs.

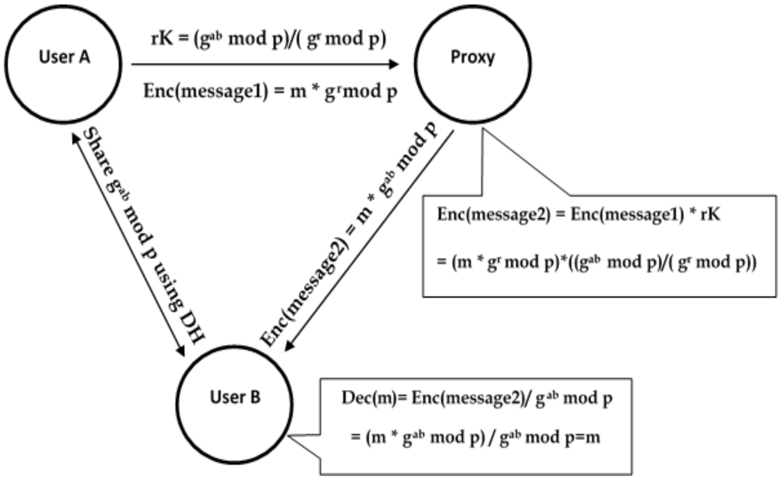

User A
rK = (g^{ab} mod p)/(g^{r} mod p)
Enc(message1) = m * g^{r}mod p
Proxy
Share g^{ab} mod p using DH
Enc(message2) = m * g^{ab} mod p
Enc(message2) = Enc(message1) * rK
= (m * g^{r} mod p)*((g^{ab} mod p)/(g^{r} mod p))
User B
Dec(m)= Enc(message2)/ g^{ab} mod p
= (m * g^{ab} mod p) / g^{ab} mod p=m

Figure 33: The proposed symmetric proxy re-encryption

5.3.2 The Proposed Security Protocol

The proposed protocol for securing the communication between the client application, the IoT devices, and the on-cloud FPGA is illustrated using the sequence diagram in Figure 34. The client application is a piece of software that is responsible for authenticating the FPGA, securely sharing keys with the FPGA, securely sending configuration bitstream and checking the configuration integrity while the configuration bitstream is running in the FPGA. On the IoT device side, the IoT device also needs to authenticate the FPGA and share a key with it. The IoT data is stored in the cloud using one key (g^{r} mod p) and the IoT device gives delegation for the authenticated FPGA to decrypt and process the data.

- The protocol is initiated by the client who sends a request for a physical resource (i.e. the FPGA) to the CP. The FPGA is assigned to one client and can receive data from multiple IoT devices. The CP assigns an FPGA for the client and sends back its identifier (ID(Fi)) (step 1 and step 2 in Figure 34),
- The client forwards the ID(Fi) to the TA which responds with the following FPGA authentication credentials; a session mask M that consists of an n-bit random number with exactly L number of 1s, hash of the corresponding L-bit number b concatenated with ID(Fi), and the FPGA's session key portion (g^b mod p) (step 3 and step 4 in Figure 34). Note that both g and p are public values with g usually being a small integer such as 2 and p being a prime number satisfying the condition $g^b \geq p$. Similarly, g^a must be $\geq p$,
- The client forwards M and its own portion of the session key, g^a mod p, to the CP and requests FPGA authentication credentials. The FPGA will use M to generate *b* using the masking circuitry (step 5 in Figure 34). The FPGA uses *b* to generate its portion of the session key, (g^b mod p), computes Hash(*b*+ID(Fi)), and sends the result back to the client (step 6 in Figure 34). The client can now authenticate the FPGA by comparing the values of Hash (*b*+ID(Fi)) and (g^b mod p) received from the TA and CP. This prevents MiM and FPGA impersonation attacks [26]. Both parties now share the symmetric session key g^{ab} mod p, completing the Ephemeral Diffie–Hellman key exchange. At this point, a and b are destroyed by the client and the FPGA, respectively. In addition, the session key will be destroyed at the end of the session to achieve the desirable security feature of Perfect Forward Secrecy (PFS),

- The client sends his/her circuit's configuration bitstream Bit(client) encrypted using g^{ab} mod p and the fixed logic on the FPGA will then decrypt it and use it to configure the FPGA through the ICAP (step 7 in Figure 34),
- The CP broadcasts the ID of the FPGA to the client's IoT device. The IoT device then sends the FPGA ID to the TA and the TA responds with a new mask (*M1*) along with the hash and the key portion of the Diffie-Hellman key exchange (g^{b1} mod p). The IoT device requests FPGA authentication by sending *M1* and its Diffie-Hellman key exchange portion (g^{a1} mod p) to the FPGA. The FPGA responds by providing the hash and the (g^{b1} mod p). The IoT device can then compare the hashes and keys portions received from both the TA and the FPGA (steps 8-12 in Figure 34). If there is a match, the session will be established. Otherwise, it will be terminated,
- The FPGA encrypts the key (b1), which is generated by the PUF circuitry, using the session key established (g^{a1b1} mod p) and sends it to the IoT device (step 13 in Figure 34). The IoT device sends the re-encryption key (rK1 = b1/g^{r} mod p) to the on-cloud proxy which in turn transforms the IoT device data that is encrypted using (g^{r} mod p) to be encrypted using b1 as in the scheme discussed in Figure 33 and sends the re-encrypted data to the FPGA (step 14 and step 15 in Figure 34). The FPGA receives M1 from the CP and regenerates b1 and decrypts the data and sends the result to the client after encrypting it using the session key (g^{ab} mod p). The value b1 is regenerated to avoid storing large number of keys thereby eliminates the need for memory resources needed to store the keys and allowing any number of IoT devices to process their data in the FPGA,

– To protect against any circuit tampering (e.g. HW Trojans or sniffing circuitry inserted on the FPGA), the client chooses a secure random value RN, encrypts it with the g^{ab} mod p and sends it to the FPGA requesting configuration readback. The Static Logic decrypts RN, reads back the FPGA configuration, hashes it with RN, encrypts with the session key, and sends it back to the client (steps 17 and 18 in Figure 34). The client can use this to validate the integrity of the FPGA. This check can be repeated any number of times (with a new RN every time to prevent replay attacks), during the operation of the client's circuit on the FPGA.

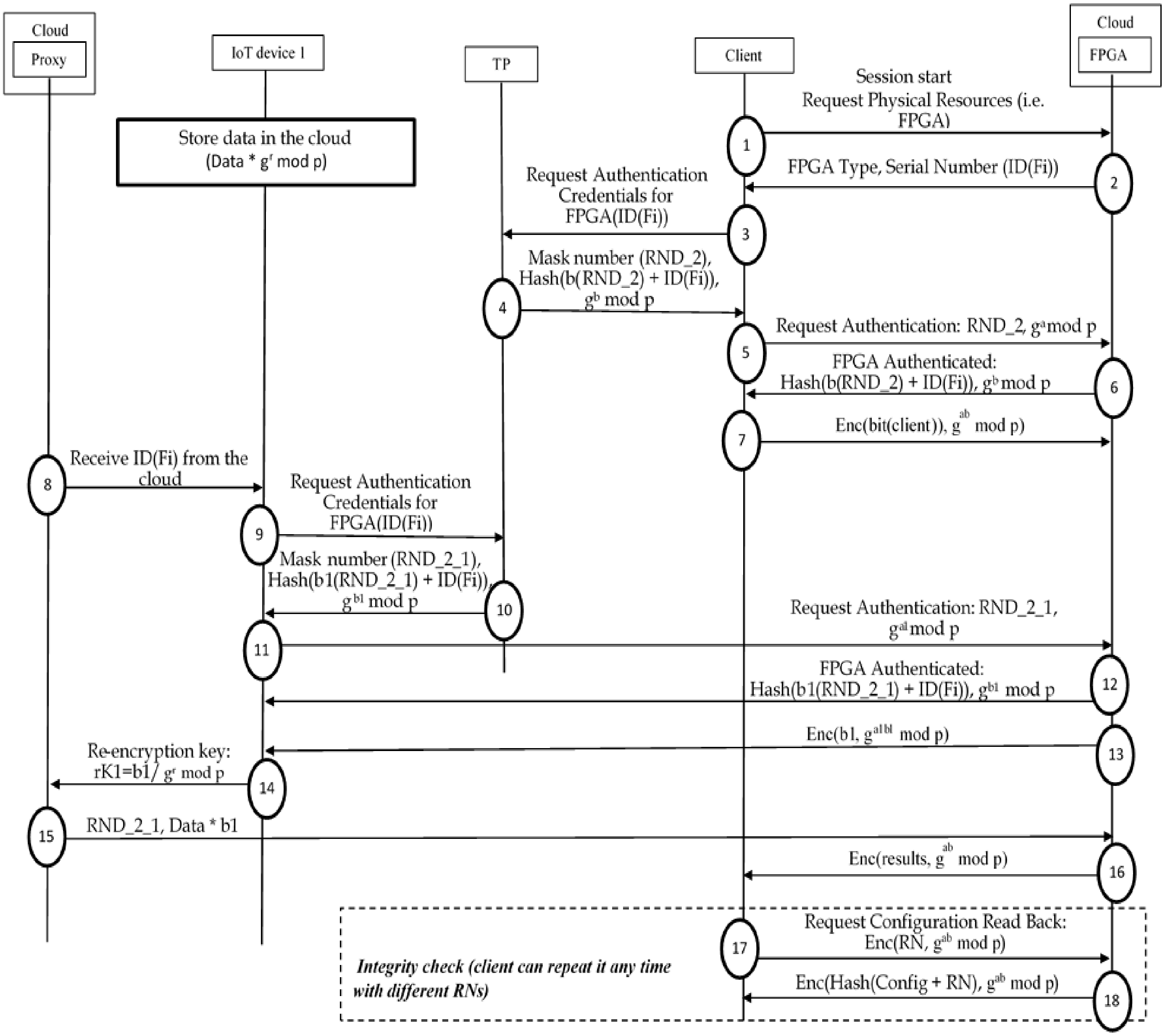

Cloud
Proxy
IoT device 1
TP
Client
Cloud
FPGA
Store data in the cloud
(Data * g^r mod p)
Session start
Request Physical Resources (i.e. FPGA)
1
FPGA Type, Serial Number (ID(Fi))
2
Request Authentication Credentials for FPGA(ID(Fi))
3
Mask number (RND_2), Hash(b(RND_2) + ID(Fi)), g^b mod p
4
Request Authentication: RND_2, g^amod p
5
FPGA Authenticated: Hash(b(RND_2) + ID(Fi)), g^bmod p
6
Enc(bit(client)), g^{ab} mod p)
7
Receive ID(Fi) from the cloud
8
Request Authentication Credentials for FPGA(ID(Fi))
9
Mask number (RND_2_1), Hash(b1(RND_2_1) + ID(Fi)), g^{b1} mod p
10
Request Authentication: RND_2_1, g^{a1}mod p
11
FPGA Authenticated: Hash(b1(RND_2_1) + ID(Fi)), g^{b1} mod p
12
Enc(b1, g^{a1b1} mod p)
13
Re-encryption key: rK1=b1/ g^r mod p
14
RND_2_1, Data * b1
15
Enc(results, g^{ab} mod p)
16
Request Configuration Read Back: Enc(RN, g^{ab} mod p)
17
Enc(Hash(Config + RN), g^{ab} mod p)
18
Integrity check (client can repeat it any time with different RNs)

Figure 34: The sequence diagram of the protocol.

The steps above are repeated for every session and M is never repeated. For step 13 in Figure 34, b1 could be XORed with a private static FPGA number and the resulted key would be sent to the IoT devices instead of b1 to avoid exposing b1 outside the FPGA. It should be noted that this scheme also supports 3rd-party provided circuit IPs (i.e. the circuit is provided by an IP vendor). In this case, to protect the circuit IPs, the IP vendor will encrypt the circuit IP(s) using a different Mask and key obtained through similar steps, and perform the integrity checks.

5.3.3 Security Analysis

This section, similar to section 4.2.6, summarizes the possible attacks of our IoT scheme and the countermeasures the scheme is providing. Table 9 summarizes most popular attacks and describes what protection mechanism our scheme can provide to prevent these attacks.

ProVerif [187] was used for automatic verification of the proposed protocol and to ensure that the protocol does not suffer from any vulnerabilities. The following assumptions were made:

- We modeled the interactions between the IoT-device and the FPGA as this also models the interactions between the client and the FPGA,
- The attacker has access to all communication channels except for private channels,
- To verify the match of the hash values received from the TA and the FPGA in the IoT device side, these values are sent to the IoT device and the FPGA. The FPGA then send the value received from the TA to the IoT device to emulate the operations of the b generation and its corresponding hash value,
- The channel between the IoT device/FPGA and the TA is private,
- The attacker is active which means that the attacker has full access to all messages and can send or replay messages in the communication channels.

Appendix C shows the ProVerif code of our proposed protocol which consists of the following parts:

- Channels involved and adversary model are in lines 3-8,

- Encryption/decryption and hash functions models are in lines 9-16,
- The TA operations are in lines 19-22,
- The IoT-device operations are in lines 23-29,
- The FPGA operations are in lines 30-34.

The results of this ProVerif code shows that the query is true; indicating that the protocol is free from vulnerabilities.

Table 9: Summery of countermeasure against most popular attacks

Attack category	Attacker	Countermeasures
Malicious proxy	malicious insider	This attack is ineffective because the data is not decrypted by the proxy and the attacker sees encrypted data only.
Proxy and IoT device collusion	malicious insider who impersonates the IoT device	If the proxy and the IoT device collude by impersonating the IoT device, the scheme is still secure since the b key is known to the attacker and different key is used every session.
Proxy and FPGA collusion	malicious insider who impersonates the FPGA	This attack is prevented by using the PUF-RN which is known to the FPGA and the TA only. Further, the private key of the IoT device is not exposed to any party, including the FPGA.
Cryptographic attacks	assumed to be a malicious insider who tries to break the cryptographic oracle and obtains the	No plaintext version of any encrypted data is ever made available to any party other than the one who generated it (i.e. any data outside the FPGA is encrypted). open key attack model is prevented using the PUF, which produces random uncorrelated numbers from which the key is generated. In addition, the steps of the protocol are repeated for every session, the session mask (*M*) is never repeated for stronger protection, and *b* is never disclosed as

	session key established between the client/IoT device and the FPGA.	a plaintext to the client or to any other party.	
Network attacks	assumed to be a malicious insider/outsider attempting to impersonate the FPGA and/or obtain sensitive data.	impersonation	Prevented as shown in Figure 18
		MiM	Prevented as shown in Figure 19
		replay	Replaying the values to be sent by the FPGA is prevented because *M* is never repeated. Integrity checking is also secured through the use of the symmetric session key, and replaying it is prevented through the use of the newly client-generated random number (*RN*).
Physical and FPGA attacks	assumed to be a malicious insider that has access to the FPGA devices in the datacenter and is trying to obtain the device secrets and the IoT sensitive data.	invasive	Damage the FPGA and any divulged secrets such as the PUF-RN are useless because it is only unique to that FPGA.
		non-invasive	All blocks of the *static logic* have constant processing time (i.e. cycles). Similarly, Power and Electromagnetic Radiation analysis attacks are mitigated due to the use of differential PUF-RN circuitry and similar techniques for the security components such as the RSA [188]; the power/electromagnetic profiles do not depend on the value of *b* or the shared key.
		semi-invasive	The required knowledge and equipment are beyond a malicious insider.

5.4 Results and Discussion

In this section, we describe the FPGA implementation of the proposed protocol and report its resource overhead as well as the PRE performance of both software and FPGA implementation.

5.4.1 FPGA Implementation

To evaluate the practicality and performance of the proposed scheme, a complete proof-of-concept prototype of an FPGA system has been implemented. A Xilinx Virtex-6 LX 550T FPGA prototyping board (with 1 Gbps Ethernet ports) was used for the prototype. For prototyping purposes, the Static Logic blocks were implemented using the FPGA's reconfigurable logic blocks. The Static Logic is made of the following components:

- A 512-bit SHA3 hashing block to support 256-bit session keys. This circuit was designed and implemented based on the Keccak sponge function reported in [155]. The design required major changes to make it routable and to pipeline it (mainly rounds steps),
- A 256-bit modular multiplier based on the interleaved modular multiplication algorithm [189],
- An OpenCore implementation of the modular exponentiation block (modexp) based on the Square-and-Multiply algorithm by McQueen [157]. The modexp was also used to implement the multiplicative modular inverse,
- The PUF as a 2048-bit register containing a random number, and the masking circuitry (as shown in Figure 16),

- An FSM, Figure 35, to control the components of the static logic.
- An Ethernet controller and a state machine to handle the data flow between the components.

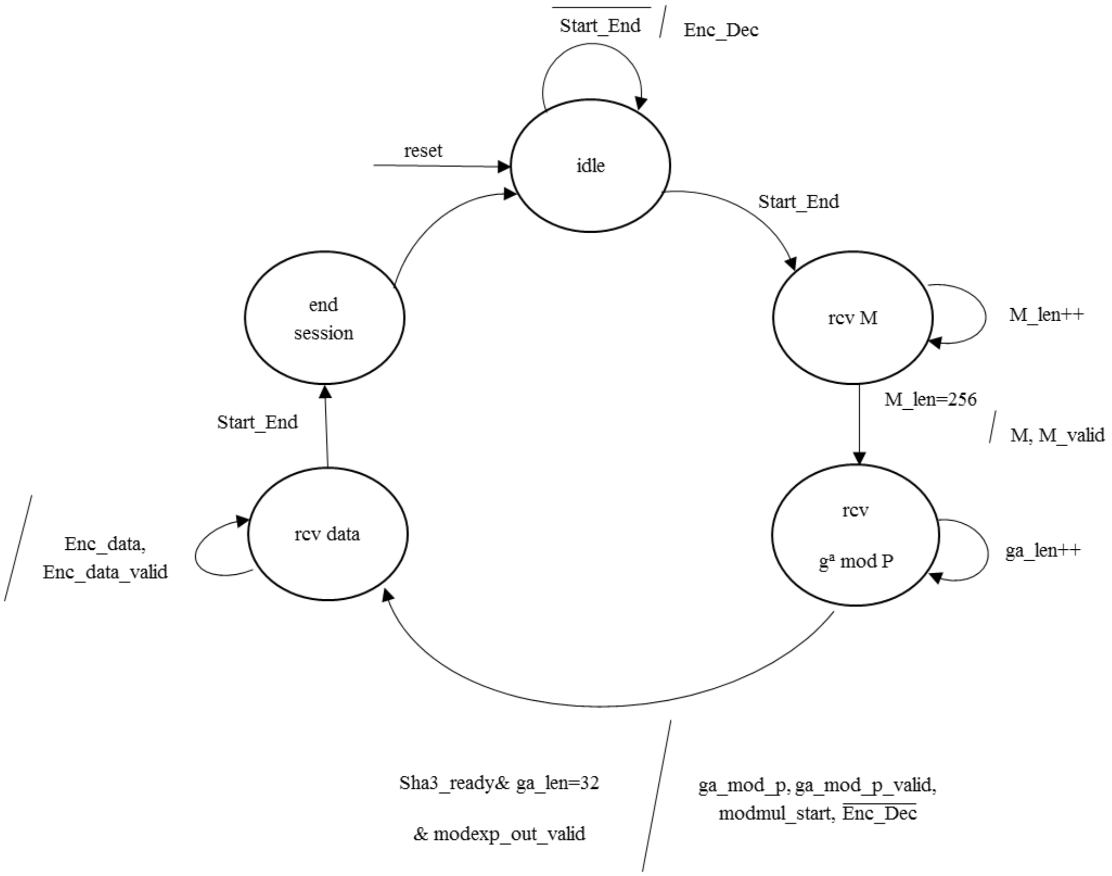


Figure 35: Main FSM of the IoT scheme

The FPGA's logic and memory utilization of the different Static Logic blocks is shown in Table 10 along with their maximum possible frequencies. These results show that even if the Static Logic components were to be implemented using the FPGA's configurable resources they would consume relatively very low resources (~5% of LUTs, ~2.8% of flip-flops, ~1.9% of the available block RAMs, and ~3.4% of the available DSP multipliers). Prior work ([158] [159] [160]) reported similar results indicating that these types of functions can be implemented very efficiently on FPGAs.

The Static Logic was also synthesized as a custom circuit to estimate its area if it was made as hard macros on the FPGA. The total gate count was 112,877 gates (total RAM and FFs count remain the same as the FPGA implementation). Based on that, and to put this into perspective, the total area of the Static Logic as custom HW macros is estimated to be 0. 0380 mm2 in a state-of-the-art 16/14 nm fabrication technology based on the International Technology Roadmap for Semiconductors (ITRS) [161]. A typical state-of-the-art FPGA would have a die area from few hundred mm2 to around 2,000 mm2 [162].

As shown in Table 10, the Static Logic synthesized on the FPGA was also relatively fast. All components used the 100 MHz FPGA board clock since that was more than enough to handle the board's 1 Gbps Ethernet traffic. The SHA3-512 achieved a throughput of 237MB/s and a latency of 27 cycles to process 64B of data. Similarly, the 256-bit modular multiplier takes 256 cycles to process 256 bits of data and can be enhanced by making multiple copies of it to work in parallel as will be discussed in Section 3.2 below. Modexp component is rarely used and it is used only at the beginning of the session and when calculating the modular multiplicative inverse of b using Fermat's little theorem as in equation (1). It takes less than 0.7ms to perform modular exponentiation for 256-bit base and exponent with the 256-bit modulus. The latency of our basic masking circuitry is 2048 cycles for the 2048 bit PUF-RN. These components can be easily operated at higher frequencies to handle higher bandwidth Ethernet links.

Table 10: Resource utilization of the Static Logic

Static Logic	LUTs	FFs	BRAMs	DSP	F_{Max} (MHz)
Full System	17,386 (5.06%)	19,443 (2.83%)	12* (1.89%)	29 (3.36%)	234.9
SHA3-512	7,573 (2.20%)	2,211 (0.32%)	0 (0.00%)	3 (0.35%)	273.9
Ethernet Controller	1,302 (0.38%)	1,045 (0.15%)	12 (1.89%)	19 (2.20%)	234.6
Enc-Dec	2,107 (0.61%)	773 (0.11%)	0 (0.00%)	0 (0.00%)	134.7
modexp	6,816 (1.98%)	3,595 (0.52%)	0 (0.00%)	0 (0.00%)	130.6
Masking circuitry	3,340 (0.68%)	4,349 (0.68%)	0 (0.00%)	2 (0.23%)	430.3
FSM	2,488 (0.72%)	2,460 (0.36%)	0 (0.00%)	2 (0.23%)	413.6

* ~ 264 Kb out of 22,752 Kb total.

$$a\,.\,a^{q-2} \equiv 1\ (mod\ q) \qquad (1)$$

Figure 36 shows the actual signals obtained from the implemented prototype using Xilinx Chipscope (a technology that allows real-time monitoring of on-FPGA buses). The capturing is done for establishing a session with an IoT device, which takes more steps than that establishing a session with the client as shown in Figure 34. Only the least 32 significant bits of each bus are displayed in Chipscope since the maximum triggers that can be shown in Chipscope is 256 bits. In addition, the signals were captured from three successive runs and the values should match the values in the example in Table 11. In the 1st run (Figure 36(a), M is first received, then g^a mod p is received. For the 2nd run, Figure 36(b), both M and g^a mod p are set as their values in the first run as in Table 11, then b is generated by the masking circuitry, then hash(b+ ID(Fi)) is computed and sent back to the client. The third run (Figure 36(c)) is triggered when encrypted data is valid thereby capturing the values of g^b mod p followed by g^{ab} mod p followed by the encrypted b that is sent to the IoT device. Figure 36(c) also shows the encrypted data and the decrypted data along with their valid signals. The encrypted data is supposed to be the same as re_data in Table 11. More snapshots of the design and its different components can be found in Appendix A.

Table 11: keys and hash values Examples

Name	Size in bits	Value
g	2	0x3
q	256	0xd0a6b524f46f5a59520d3efcba360545d911e748700ff141b7414405bcd22c0b
r	256	0x1b1ba9a04575d309395ed00546339621904dafe5094ed826d081af26407f00a2
g^r	256	0x1302d7d599d1ec79d677e7eee28c6b565841563b17f6f3146aebc36a6382d841
a	256	0x14bb28715d971d180f7055e2098e1a8a2ff67c4090afc649dc69f2424f62ccec
b	256	0xa78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4
g^a	256	0x4b058bb3c58c38662bb2b8eb58534a24cba7e5194cedcb61c1f9cf5b0d890e78
g^b	256	0xf343a2e3522bba046a7ded8510fd2d17b6ac9faa0cb96f346a21b9668bd3164
g^{ab}	256	0x6c528c1cef10ae5184e3f2a0f752b5fbb004928e80811282233b9847d3212e99
data	256	0x1111111111111111111111111111111111111111111111111111111111111111
g^r*data	256	0x7d36a1a577e3b7aedc9a8fcafc0baa6628bf27ed613db443cb0e1984cbf6150
rK	256	0x6e81c5e894ddb9371b6833cca4a8c39a96f79159d38ea7eee941cd9014063bb8
re_data	256	0x78c9acbd53051363d7cb66ff097e2bf568ec52055b878311360594f27612d5ea
Enc_b	256	0x97b894cce3b431462322142fec06a5d517b139ce288d92195336247f7ccf7c4f
ID	64	0xffffffffffffffff
SHA3(b+ID)	512	0xa5ecac8593f8561a7475e729ea89aa4f118b8472260356587a3aaa804667a332791fcb9bc8f7b1fa429286925c6e7a3abf7e22c7381f624d4046afd49a96ea12

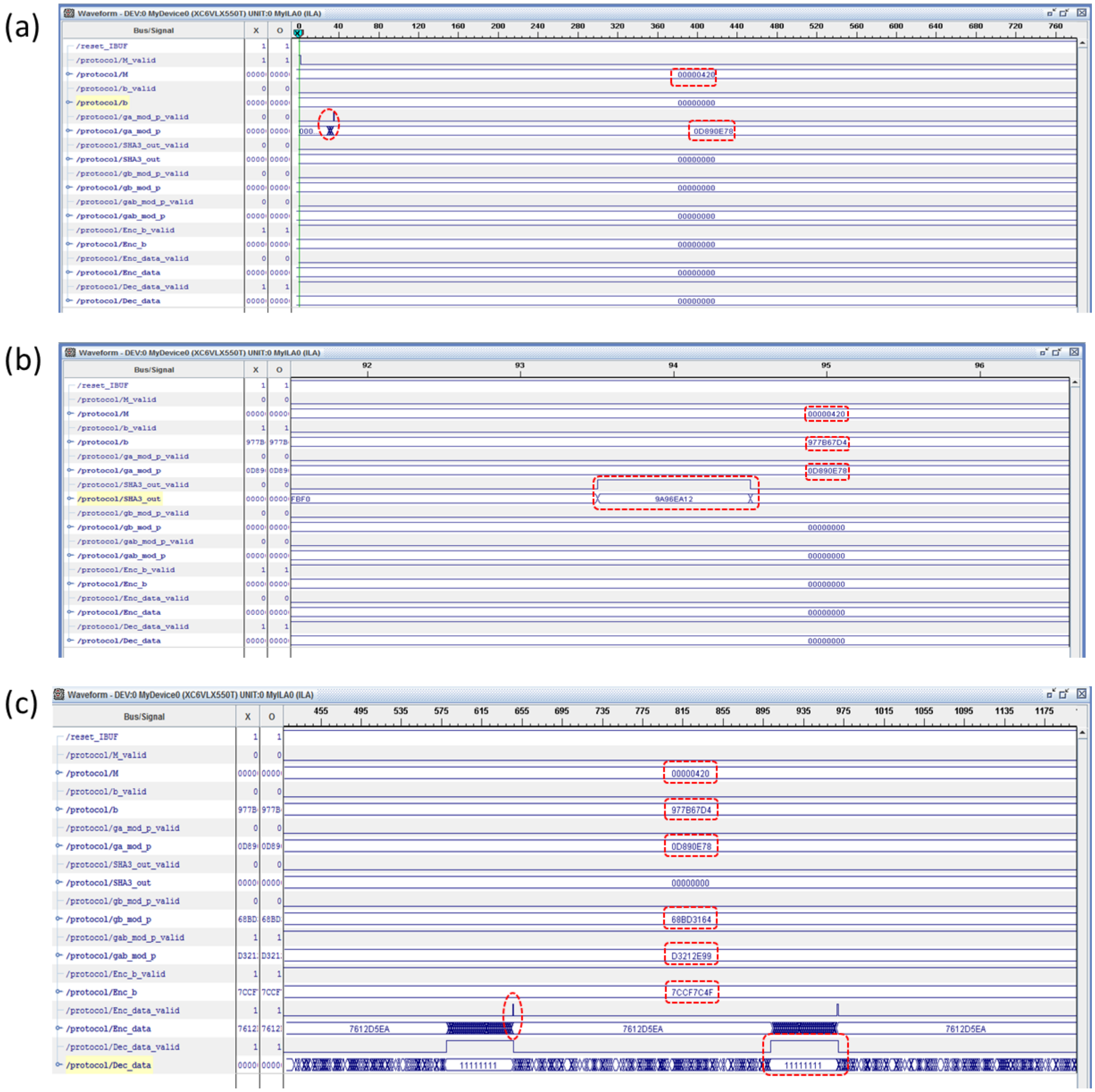


Figure 36: Chipscope screenshots showing the operations of the implemented Static Logic on the FPGA. (a) Receiving M, ga mod p, (b) Producing b and the hash values, (c) Producing gb mod p, gab mod p, encrypted b and receiving the encrypted data and decrypting it.

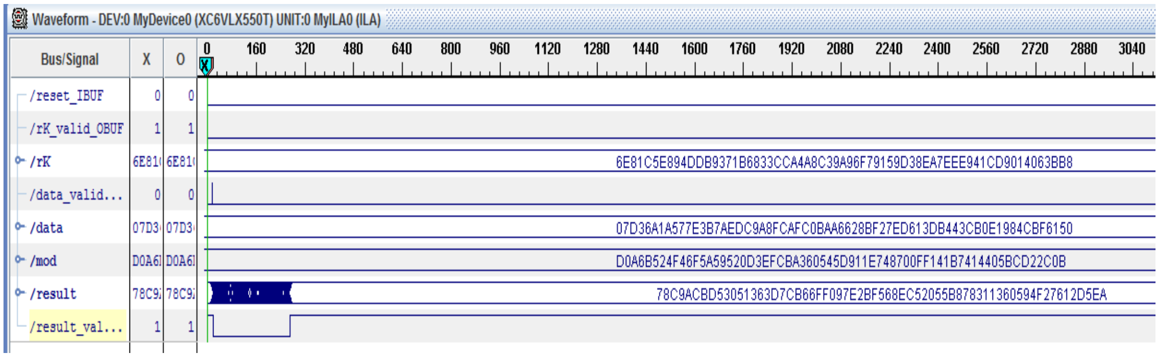
5.4.2 PRE Implementation

The PRE was also implemented in another Virtex 6 LX 550T FPGA. The proxy design can be installed once in an untrusted manner since the data is not decrypted in the FPGA. Our FPGA-based proxy consists of an Ethernet controller that receives TCP packets, 256-bit modular multiplier, and an FSM to receive the rK key and accumulate the data into 256 bits chunks to be multiplied by the rK. The components of the FPGA along with their maximum proxy are shown in Table 12. The synthesis results suggest that our proxy can be implemented in low-cost FPGAs and there could be many multipliers working in parallel. For a 1 Gbps Ethernet link, 10 bits can be processed every cycle. For this reason, we decided to make a BRAMS for buffering the data that is received/sent from/to Ethernet link and 10 multipliers to work in parallel such that 250 bits of the product result are outputted every cycle, ignoring the latency off filling the BRAMs and computing the first product result at the beginning.

Table 12: FPGA resource usage by the PRE Logic.

PRE Logic	LUTs	FFs	BRAMs	DSP	F_{Max} (MHz)
Full System	21068 (6.13%)	8534 (1.24%)	12 (1.89%)	21 (2.43%)	135
ModMult x 10	2,107 (0.61%)	773 (0.11%)	0 (0.00%)	0 (0.00%)	134.7
Ethernet Controller	1,302 (0.38%)	1,045 (0.15%)	12 (1.89%)	19 (2.20%)	234.6
FSM	266 (0.07%)	260 (0.03%)	0 (0.00%)	2 (0.23%)	274.6

Figure 37 shows an example of our implemented proxy for the result and signals of one multiplier. The image is triggered when the rk_valid signal goes high and it shows the data_valid signal goes high. The last signal of the image is the result_valid signal and it goes high when the result is ready or when in an ideal state. The rK is as in the example in Table 11 and the data is what is received from the IoT device (g^r mod p*data).



Waveform - DEV:0 MyDevice0 (XC6VLX550T) UNIT:0 MyILA0 (ILA)

Bus/Signal	X	O	
/reset_IBUF	0	0	
/rK_valid_OBUF	1	1	
/rK	6E81(	6E81(	6E81C5E894DDB9371B6833CCA4A8C39A96F79159D38EA7EEE941CD9014063BB8
/data_valid...	0	0	
/data	07D3(	07D3(	07D36A1A577E3B7AEDC9A8FCAFC0BAA6628BF27ED613DB443CB0E1984CBF6150
/mod	D0A6I	D0A6I	D0A6B524F46F5A59520D3EFCBA360545D911E748700FF141B7414405BCD22C0B
/result	78C9/	78C9/	78C9ACBD53051363D7CB66FF097E2BF568EC52055B878311360594F27612D5EA
/result_val...	1	1	

Figure 37: Chipscope image showing the operations of the PRE

5.4.3 Performance Evaluation

Our proxy re-encryption can be seen as a packet processer since nothing more receiving the data and outputting the converted result is required. The natural way for implementing packet processors is the FPGAs as explained in more details in [190].

To evaluate the performance of our FPGA implementation of the proxy, we compare it with a software-based version implemented in python 2.7 (the script can be found in Appendix D). We used a workstation with Intel Xeon CPU with 8 core 3.20GHz, 23.5 GB of memory, 2 TB of disk, and 64-bit Ubuntu 14.04 OS. To evaluate the software performance, two experiments were carried out. The first experiment is to make the data available for the SW in arrays which means the data is entirely in the memory. For precise measurement of the time of the SW version, the measurements were repeated 10000 times and the average of 10 runs was taken; producing the 1000 measurements in Figure 40 for 1Mb of data. The minimum time was set to be the actual value and the maximum time was set to be the experimental value and the percentage error was calculated and found to be 2.76% only.

The second experiment was carried out by measuring the performance of the SW by reading the data from the disk to model the actual performance as it is the case when the SW runs in the cloud. Figure 42 shows the trend line of 1Mb of data. The percentage error was found to be 3.80% only.

The FPGA implementation makes use of the data initialized in the BRAMs. Based on these setups, Figure 38 shows the time it takes in seconds for both experiments and the for the FPGA and Figure 39 and Figure 41 depict the speedup of the FPGA implementation over the SW implementations. Reading the data from the disk is about 1.76x slower compared to reading the data from the memory. It can be noticed from Figure 42 that there are some small jumps due to accessing the disk. The speedup is shown in Figure 39 and Figure 41 for both experiments. The FPGA implementation is, on average, 5.8 times faster than the SW implementation while the data in memory and about 10.26 times faster than the SW implementation when the data is read from the disk. Given the speedup obtained, if the Ethernet link speed is 10 Gbps instead of 1 Gbps, the speedup will be about 58 when the data is read from the memory and about 102.6 when the data is read from the disk.

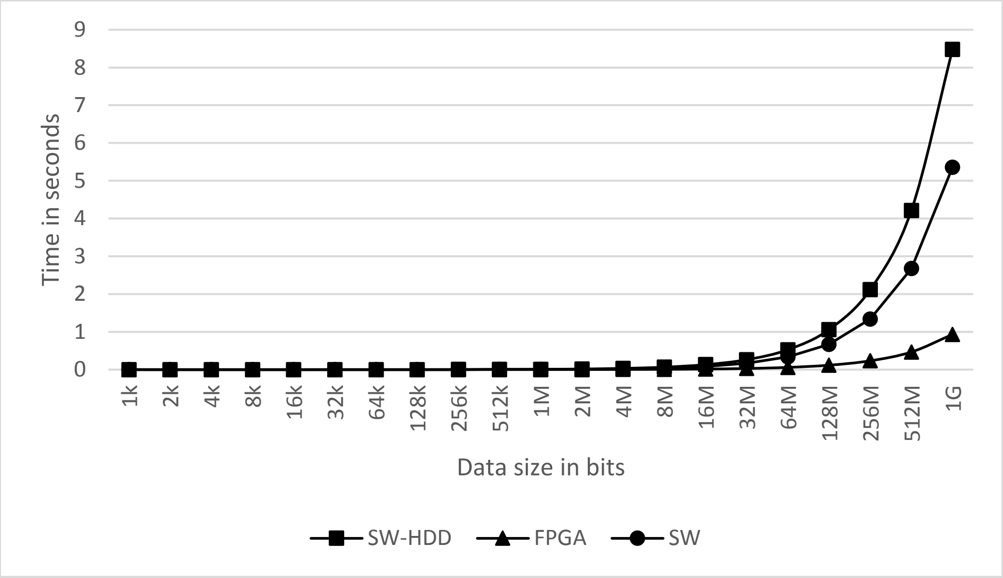


Figure 38: Time comparison of the PRE FPGA implementation and the SW PRE. In the SW implementation, the data is read from the memory and SW-HDD means that the data is read from the disk.

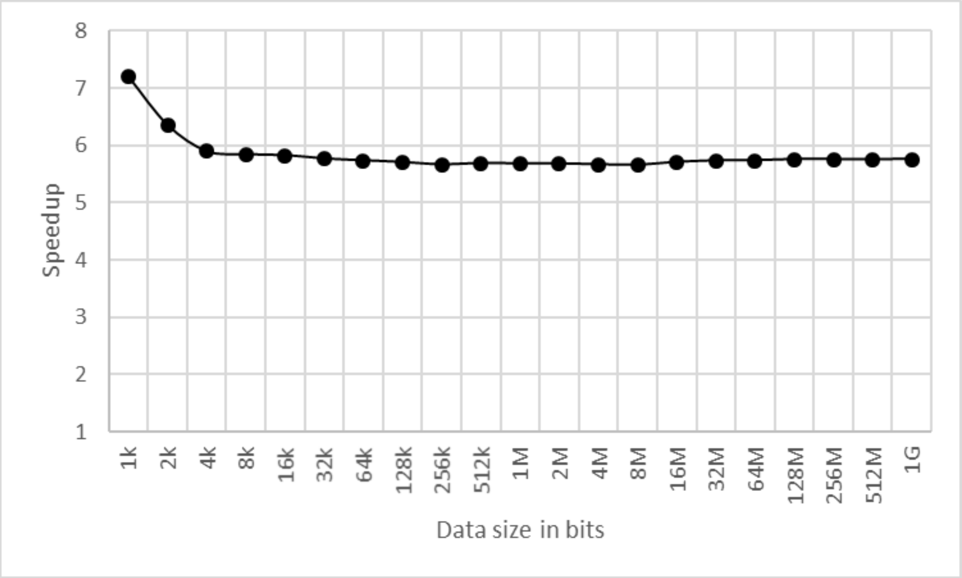


Figure 39: The speedup obtained by our PRE FPGA implementation over PRE SW implementation. The data is read from the memory.

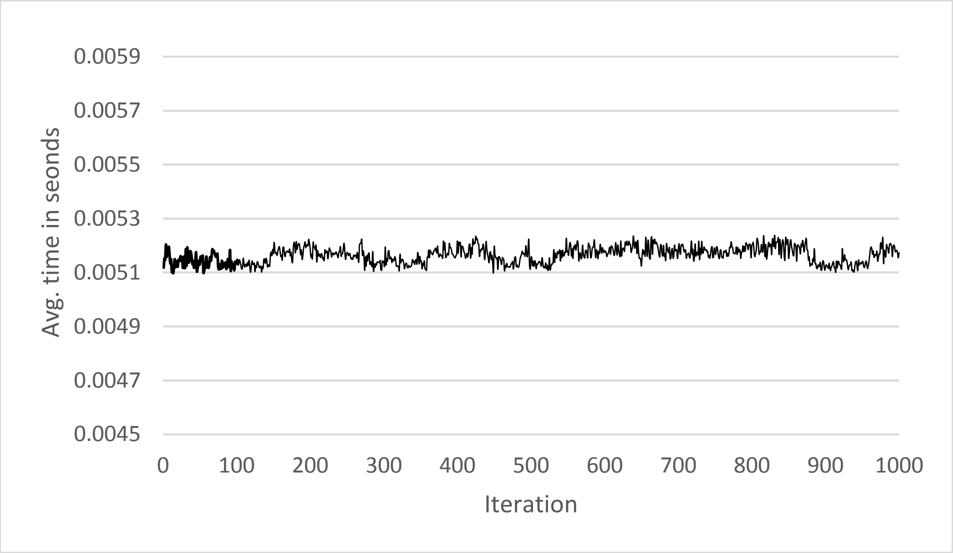


Figure 40: The time of the python PRE over 1000 runs. The data is read from the memory.

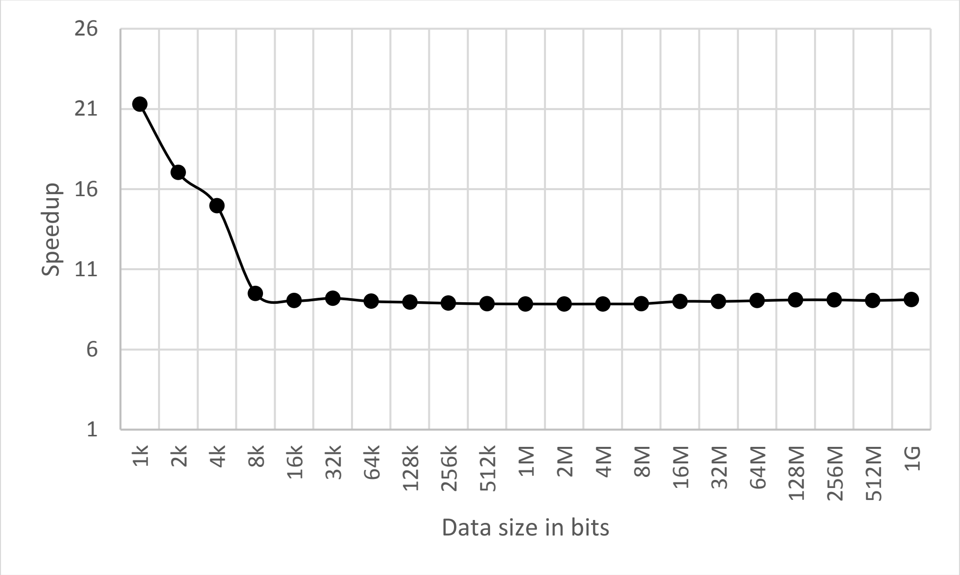


Figure 41: The speedup obtained by our PRE FPGA implementation over PRE SW implementation. The data is read from the disk.

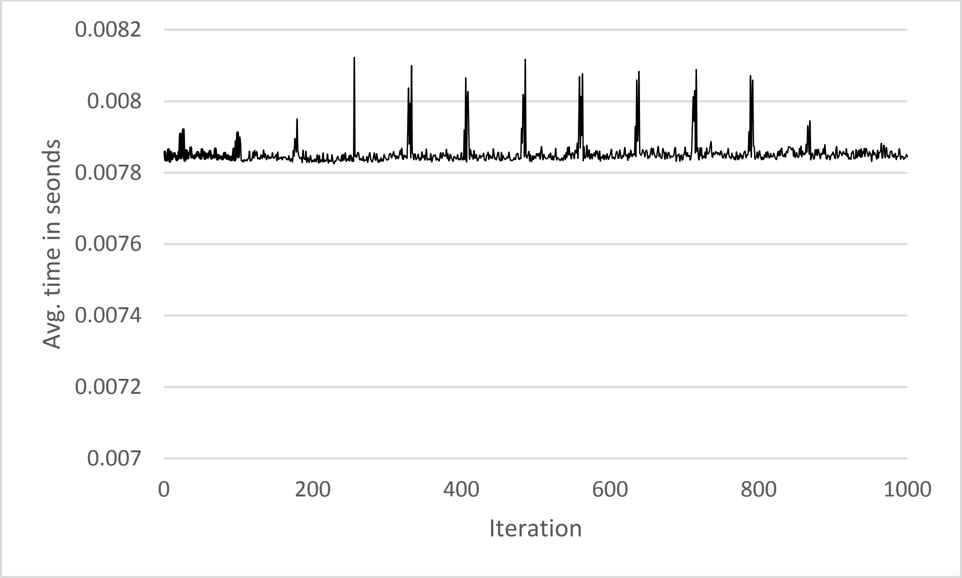


Figure 42: The time of the python PRE over 1000 runs. The data is read from the disk.

5.5 Conclusions

In this work, a new FPGA-based scheme for securing IoT data in clouds is proposed, including a symmetric proxy re-encryption. It was shown that the proposed protocol for establishing a secure session on a cloud's FPGA provides strong protection against various types of attack. A complete proof-of-concept prototype implementation of the scheme showed that it is feasible even with existing FPGAs, simple to implement, efficient in terms of resource utilization and suites the publish/subscribe model. The proposed scheme achieves perfect forward secrecy, provides authentication of the on-cloud FPGAs by the clients and integrity checking of client configuration to prevent any modification and/or other FPGA related attacks such as reverse-engineering and cloning.

CHAPTER

CONCLUSION AND FUTURE WORK

In this dissertation, we studied the existing techniques for the problem of securing client data in the cloud. Based on this, we proposed a novel scheme based on FPGAs to tackle this problem. We developed all the SW/HW components of the scheme and proposed protocols to securely communicate with the on-cloud FPGAs. We also showed that our scheme can be easily integrated in the cloud as a cloud resource with a boot time that is 15x faster than booting a conventional VM. We also showed that in terms of performance, our solution is faster and more secure than existing solutions such as Intel SGX.

Moreover, we extend the space of our solutions to more challenging security situations such as securing IoT data in the cloud. The results depicted that our scheme for handling IoT data is efficient in terms of FPGA resource overhead and performance. We also handle securely the transformation of encrypted IoT data in the cloud by proposing a symmetric proxy re-encryption. Our proxy re-encryption performance was reported and suggested that it is best suited for FPGAs to perform the transformation, which is at least 6x faster than Xeon machines when using the 1G Ethernet and is at least 60x faster when using the 10G Ethernet.

This work can be extended in many ways. It opens huge opportunities for many contributions. An obvious extension is using our work for securing client data for virtualized FPGAs. Another extension is IP protection in the cloud environment. Further, exploiting FPGAs for application that can be parallelized such that the benefits of security and speed can be combined by using our schemes. For IoT and smart grids, our solutions can be extended to function as a web services in the FPGAs and exhibit the machine-to-machine communication, in which FPGAs are well suited for and is expected to dominate in this field since the FPGAs brings the flexibility for bringing the computation closer to the edge allowing the IoT data sent to the cloud to reduce in size.

REFERENCES

[1] I. T. Report, “The 2013 Vormetric Insider Threat Report,” 2013. [Online]. Available: ljo.es/TheVormetricInsiderThreatReportOct2013.pdf.

[2] C. I. T. Center, “How bad is the insider threat?,” 2013. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=58738.

[3] T. Ristenpart and E. Tromer, “Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds,” in *Proceedings of the 16th ACM conference on Computer and communications security. ACM*, 2009, pp. 199–212.

[4] M. Arrington., “In our box: Hundreds of confidential twitter documents,” 2009. [Online]. Available: http://techcrunch.com/2009/07/14/in-our-inbox-hundreds-of-confidential-twitter-documents.

[5] P. S. Voss, “Towards unique performance using FPGAs in modern communication , data processing and sensor systems,” in *Signal Processing and Integrated Networks (SPIN), 3rd International Conference on. IEEE*, 2016, pp. 1–4.

[6] XenSource Inc, “xen.” [Online]. Available: www.xensource.com.

[7] KVM, “Kernal Based Virtual Machine.” [Online]. Available: www.linux-kvm.org/page/.

[8] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: State-of-the-art and research challenges,” *J. Internet Serv. Appl.*, vol. 1, no. 1, pp. 7–18, 2010.

[9] B. Jennings and R. Stadler, “Resource Management in Clouds : Survey and Research Challenges,” *J. Netw. Syst. Manag. 23.3*, vol. 1, pp. 567–619, 2015.

[10] P. Swierczynski, M. Fyrbiak, C. Paar, C. Huriaux, and R. Tessier, “Protecting against Cryptographic Trojans in FPGAs,” in *Field-Programmable Custom Computing Machines (FCCM), IEEE 23rd Annual International Symposium on. IEEE*, 2015.

[11] J. King and K. Lauerman, “ARP Poisoning Attack and Mitigation Techniques,” *Layer 2 Attacks Mitig. Tech. Cisco Catal. 6500 Ser.*, 2010.

[12] S. Kumar, “PING attack - How bad is it?,” *Comput. Secur.*, vol. 25, no. 5, pp. 332–337, 2006.

[13] S. Kumar, “Smurf-based Distributed Denial of Service (DDoS) attack amplification in internet,” in *Second International Conference on Internet Monitoring and Protection, ICIMP 2007*, 2007.

[14] A. Lavorgna, *Loukas, George (2015). Cyber-Physical Attacks. A growing Invisible Threat*, no. No.2. 2016.

[15] D. E. Rob, *Cryptography and Data Security*. 1982.

[16] S. P. Skorobogatov, “Semi-invasive attacks-a new approach to hardware security analysis,” *Tech. report, Univ. Cambridge, Comput. Lab.*, no. 630, p. 144, 2005.

[17] R. S. Chakraborty, I. Saha, A. Palchaudhuri, and G. K. Naik, “Hardware trojan insertion by direct modification of FPGA configuration bitstream,” *IEEE Des. Test*, vol. 30, no. 2, pp. 45–54, 2013.

[18] S. Mal-Sarkar, Sanchita and Krishna, Aswin and Ghosh, Anandaroop and Bhunia, “Hardware Trojan Attacks in FPGA Devices: Threat Analysis and Effective Counter Measures,” *Proc. 24th Ed. Gt. Lakes Symp. VLSI*, pp. 287–292, 2014.

[19] E. Peeters, “Attacks with FPGA Experiments,” *Ches 2005*, pp. 309–323, 2005.

[20] A. J. Duncan, S. Creese, and M. Goldsmith, “Insider attacks in cloud computing,” in *Proc. of the 11th IEEE Int. Conference on Trust, Security and Privacy in Computing and Communications, TrustCom-2012 - 11th IEEE Int. Conference on Ubiquitous Computing and Communications, IUCC-2012*, 2012, pp. 857–862.

[21] M.-D. Nguyen, N.-T. Chau, S. Jung, and S. Jung, “A Demonstration of Malicious Insider Attacks inside Cloud IaaS Vendor,” *Int. J. Inf. Educ. Technol.*, vol. 4, no. 6, pp. 483–486, 2014.

[22] X. Wan, Z. T. Xiao, and Y. Ren, “Building trust into cloud computing using virtualization of TPM,” in *Proceedings - 2012 4th International Conference on Multimedia and Security, MINES 2012*, 2012, pp. 59–63.

[23] M. Bamiah, S. Brohi, S. Chuprat, and M. N. Brohi, “Cloud implementation security challenges,” in *Proceedings of 2012 International Conference on Cloud Computing Technologies, Applications and Management, ICCCTAM 2012*, 2012, pp. 174–178.

[24] A. A. Soofi, Khan, M.Irfan, and Fazal-E-Amin, “Encryption Techniques for Cloud Data Confidentiality,” *Int. J. Grid Distrib. Comput.*, vol. 7, pp. 11–20, 2014.

[25] C. Computing, “A Survey of Cryptographic based Security Algorithms for,” *HCTL Open Int. J. Technol. Innov. Res.*, vol. 8, no. March, pp. 1–17, 2014.

[26] R. C. Merkle, “Secure communications over insecure channels,” *Communications of the ACM*, vol. 21, no. 4. pp. 294–299, 1978.

[27] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Trans. Inf. Theory*, vol. 22, no. 6, 1976.

[28] OpenStack.com, “OpenStack Operations Guide,” *OpenStack.com*, 2013. [Online].

Available: /www.openstack.org.

[29] Chris Mitchell, *Trusted Computing*, 1st ed. Institution of Engineering and Technology, 2005.

[30] R. Stallman, “Can You Trust Your Computer?,” 2013. [Online]. Available: https://www.gnu.org/philosophy/can-you-trust.en.html.

[31] R. Anderson, “`Trusted Computing’ Frequently Asked Questions: TC / TCG / LaGrande / NGSCB / Longhorn / Palladium / TCPA Version 1.1,” 2007. [Online]. Available: http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html.

[32] L. Nkosi, P. Tarwireyi, and M. O. Adigun, “Insider threat detection model for the cloud,” *2013 Inf. Secur. South Africa - Proc. ISSA 2013 Conf.*, vol. 62500001, 2013.

[33] V. Costan and S. Devadas, “Security challenges and opportunities in adaptive and reconfigurable hardware,” *2011 IEEE Int. Symp. Hardware-Oriented Secur. Trust*, pp. 1–5, 2011.

[34] R. Wojtczuk and J. Rutkowska, “Attacking Intel Trusted Execution Technology,” *Bios*, pp. 1–6, 2009.

[35] L. Duflot, D. Etiemble, and O. Grumelard, “Using CPU System Management Mode to Circumvent Operating System Security Functions,” *CanSecWest 2006*, 2006.

[36] S. Embleton, S. Sparks, and C. C. Zou, “SMM rootkit: A new breed of OS independent malware,” *Secur. Commun. Networks*, vol. 6, no. 12, pp. 1590–1605, 2013.

[37] R. W. Joanna Rutkowska, “Preventing and detecting xen hypervisor subversions.”

Blackhat Briefings USA, 2008.

[38] F. Wecherowski, “A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers,” *Phrack*, vol. 0x0d, no. 0x42, 2009.

[39] R. Wojtczuk and J. Rutkowska, “Attacking SMM Memory via Intel ® CPU Cache Poisoning,” *Invisible Things Lab*, 2008. [Online]. Available: http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf.

[40] V. Costan and S. Devadas, “Intel SGX Explained,” *Cryptol. ePrint Arch. Rep. 2016/086*, 2016.

[41] I. Corporation, “Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3C: System Programming Guide, Part 3,” *Intel*, 2010. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf.

[42] R. Maes and I. Verbauwhede, “Physically unclonable functions: A study on the state of the art and future research directions,” in *Towards hardware-intrinsic security, information security and cryptography*, 2010, pp. 3–37.

[43] P. Tuyls, G.-J. Schrijen, B. Škorić, J. van Geloven, N. Verhaegh, and R. Wolters, “Read-Proof Hardware from Protective Coatings,” in *Cryptographic Hardware and Embedded Systems (CHES)*, 2006, pp. 369–383.

[44] D. Lim, J. W. Lee, B. Gassend, G. E. Suh, M. Van Dijk, and S. Devadas, “Extracting secret keys from integrated circuits,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, no. 10, pp. 1200–1205, 2005.

[45] G. E. Suh and S. Devadas, “Physical Unclonable Functions for Device Authentications and Secret Key Generation,” *44th ACM/IEEE Des. Autom. Conf.*,

pp. 9–14, 2007.

[46] D. Suzuki and K. Shimizu, “The glitch PUF: A new delay-PUF architecture exploiting glitch shapes,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010, vol. 6225 LNCS, pp. 366–382.

[47] S. S. Kumar, J. Guajardo, R. Maes, G. J. Schrijen, and P. Tuyls, “The Butterfly PUF protecting IP on every FPGA,” in *2008 IEEE International Workshop on Hardware-Oriented Security and Trust, HOST*, 2008, pp. 67–70.

[48] D. E. Holcomb, W. P. Burleson, and K. Fu, “Power-Up SRAM state as an identifying fingerprint and source of true random numbers,” *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1198–1210, 2009.

[49] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, “FPGA Intrinsic PUFs and Their Use for IP Protection,” *Lect. Notes Comput. Sci.*, vol. 4727, pp. 63–80, 2007.

[50] A. Maiti and P. Schaumont, “Improved ring oscillator PUF: An FPGA-friendly secure primitive,” *J. Cryptol.*, vol. 24, no. 2, pp. 375–397, 2011.

[51] F. Kodýtek, R. Lórencz, and J. Buček, “Improved ring oscillator PUF on FPGA and its properties,” *Microprocessors and Microsystems*, 2015.

[52] X. Xin, J. P. Kaps, and K. Gaj, “A configurable ring-oscillator-based PUF for Xilinx FPGAs,” in *Proceedings - 2011 14th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD 2011*, 2011, pp. 651–657.

[53] H. Yu, P. H. W. Leong, and Q. Xu, “An FPGA chip identification generator using configurable ring oscillators,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 20, no. 12, pp. 2198–2207, 2012.

[54] J. H. Anderson, “A PUF design for secure FPGA-based embedded systems,” *Proc. Asia South Pacific Des. Autom. Conf. ASP-DAC*, pp. 1–6, 2010.

[55] A. Wild and T. Guneysu, “Enabling SRAM-PUFs on xilinx FPGAs,” in *Conference Digest - 24th International Conference on Field Programmable Logic and Applications, FPL 2014*, 2014.

[56] Altera, “STRATIX 10 FPGA AND SOC.” [Online]. Available: https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html.

[57] J. Melorose, R. Perroy, and S. Careas, “PUF based FPGAs for Hardware Security and Trust,” 2015.

[58] M. Mambo and E. Okamoto, “Proxy cryptosystems: Delegation of the power to decrypt ciphertexts,” *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E80–A, no. 1, pp. 54–62, 1997.

[59] M. Blaze, G. Bleumer, and M. Strauss, “Divertible protocols and atomic proxy cryptography,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1998, vol. 1403, pp. 127–144.

[60] G. Ateniese, K. Fu, M. Green, and S. Hohenberger, “Improved proxy re-encryption schemes with applications to secure distributed storage,” *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 1, pp. 1–30, 2006.

[61] M. Green and G. Ateniese, “Identity-Based Proxy Re-encryption,” in *Applied Cryptography and Network Security*, 2007, vol. 4521 LNCS, pp. 288–306.

[62] C. Sur, C. D. Jung, Y. Park, and K. H. Rhee, “Chosen-ciphertext secure certificateless proxy re-encryption,” in *Lecture Notes in Computer Science*

(including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2010, vol. 6109 LNCS, pp. 214–232.

[63] C. K. Chu, J. Weng, S. S. M. Chow, J. Zhou, and R. H. Deng, "Conditional proxy broadcast re-encryption," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2009, vol. 5594 LNCS, pp. 327–342.

[64] G. Ateniese, K. Benson, and S. Hohenberger, "Key-private proxy re-encryption," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2009, vol. 5473, pp. 279–294.

[65] A. Syalim, T. Nishide, and K. Sakurai, "Realizing proxy re-encryption in the symmetric world," in *Communications in Computer and Information Science*, 2011, vol. 251 CCIS, no. PART 1, pp. 259–274.

[66] R. L. Rivest, "All-or-Nothing Encryption and the Package Transform," *Lect. Notes Comput. Sci.*, vol. 1267, no. Chapter 19, pp. 210–218, 1997.

[67] P. Ferrie, "Attacks on Virtual Machine Emulators," *Symantec Technology Exchange*, 2007. [Online]. Available: https://pdfs.semanticscholar.org/a72d/98d5a478efa62383a63862fc07dba831c8a5.pdf.

[68] J. R., "How to Detect VMM Using (Almost) One CPU Instruction," 2004. [Online]. Available: http://www.securiteam.com/securityreviews/6Z00H20BQS.html.

[69] Tobias K, "Scoopy Doo - VMware Fingerprint Suite," 2013. [Online]. Available:

www.trapkit.de/research/vmm/scoopydoo/index.html.

[70] L. Turnbull and J. Shropshire, “Breakpoints: An analysis of potential hypervisor attack vectors,” in *Conference Proceedings - IEEE SOUTHEASTCON*, 2013.

[71] N. Elhage, “Virtunoid: Breaking out of KVM.” Black Hat USA, 2011.

[72] K. Kortchinsky, “Cloudburst—a vmware guest to host escape story.” Black Hat USA, 2009.

[73] R. Wojtczuk, “Subverting the Xen hypervisor.” Black Hat USA, 2008.

[74] S. Ali and A. Alradha, “Protect Sensitive Data in Public Cloud from an Theft Attack and detect Abnormal Client Behavior,” *Int. J. Eng. Sci. Comput.*, no. May, pp. 552–556, 2014.

[75] R. Pawar, H. Bhapkar, S. Domal, A. Bankar, and V. Dudhale, “Reducing Inner Data Stealing Using Bogus Information Attacks in the Cloud Computing,” *Int. J. Innov. Technol. Explor. Eng.*, no. 11, pp. 37–39, 2014.

[76] C. Science and M. Studies, “Securing user data on cloud using Fog computing and Decoy technique,” *Int. J. Adv. Res. Comput. Sci. Manag. Stud.*, vol. 2, no. 10, pp. 104–110, 2014.

[77] S. J. Stolfo, M. Ben Salem, and A. D. Keromytis, “Fog computing: Mitigating insider data theft attacks in the cloud,” *Proc. - IEEE CS Secur. Priv. Work. SPW 2012*, pp. 125–128, 2012.

[78] N. J. King and V. T. Raja, “Protecting the privacy and security of sensitive customer data in the cloud,” *Comput. Law Secur. Rev.*, vol. 28, no. 3, pp. 308–319, 2012.

[79] L. Rasmusson and M. Aslam, “Protecting private data in the cloud,” in *CLOSER*

2012 - Proceedings of the 2nd International Conference on Cloud Computing and Services Science, 2012, pp. 5–12.

[80] V. Sravan and K. Maddineni, “Security Techniques for Protecting Data in Cloud Computing,” *Master Thesis*, no. November, pp. 1–75, 2011.

[81] J. Kaplan, C. Rezek, and K. Sprague, “Protecting information in the cloud,” *McKinsey Quarterly*, 2013. [Online]. Available: http://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/protecting-information-in-the-cloud.

[82] S. S. Yau and H. G. An, “Protection of Users’ Data Confidentiality in Cloud Computing,” in *Proceedings of the Second Asia-Pacific Symposium on Internetware*, 2010, pp. 1–6.

[83] S. K. Sood, “A combined approach to ensure data security in cloud computing,” *J. Netw. Comput. Appl.*, vol. 35, no. 6, pp. 1831–1838, 2012.

[84] H. Wang, Z. Wang, and J. Domingo-Ferrer, “Anonymous and secure aggregation scheme in fog-based public cloud computing,” *Future Generation Computer Systems*, 2016.

[85] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog Computing and Its Role in the Internet of Things,” *Proc. first Ed. MCC Work. Mob. cloud Comput.*, pp. 13–16, 2012.

[86] S. M., V. Patel, H. D., and N. Lakshmanan, “A Hybrid Protocol to Secure the Cloud from Insider Threats,” in *2014 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2014, pp. 1–5.

[87] M. B. B and S. M. S. Bhanu, “Analyzing User Behavior Using KeyStroke

Dynamics to Protect Cloud from Malicious Insiders," in *IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2014, pp. 1–8.

[88] L. Nkosi, P. Tarwireyi, and M. O. Adigun, "Detecting a malicious insider in the cloud environment using sequential rule mining," *IEEE Int. Conf. Adapt. Sci. Technol. ICAST*, 2013.

[89] Q. Yaseen and B. Panda, "Tackling insider threat in cloud relational databases," *Proc. - 2012 IEEE/ACM 5th Int. Conf. Util. Cloud Comput. UCC 2012*, pp. 215–218, 2012.

[90] B. M. Babu and M. S. Bhanu, "Prevention of Insider Attacks by Integrating Behavior Analysis with Risk based Access Control Model to Protect Cloud," in *Procedia Computer Science*, 2015, vol. 54, pp. 157–166.

[91] Q. Althebyan, R. Mohawesh, Q. Yaseen, and Y. Jararweh, "Mitigating insider threats in a cloud using a knowledgebase approach while maintaining data availability," in *2015 10th International Conference for Internet Technology and Secured Transactions, ICITST 2015*, 2016, pp. 226–231.

[92] Q. Yaseen, Q. Althebyan, B. Panda, and Y. Jararweh, "Mitigating insider threat in cloud relational databases," *Secur. Commun. Networks*, vol. 9, no. 10, pp. 1132–1145, 2016.

[93] C. Gentry, "a Fully Homomorphic Encryption Scheme," *PhD Thesis*, no. September, pp. 1–209, 2009.

[94] V. Vaikuntanathan, "Computing blindfolded: New developments in fully homomorphic encryption," in *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, 2011, pp. 5–16.

[95] K. Eguro and R. Venkatesan, “FPGAs for trusted cloud computing,” in *Proceedings - 22nd International Conference on Field Programmable Logic and Applications, FPL 2012*, 2012, pp. 63–70.

[96] A. Moradi, M. Kasper, and C. Paar, “Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism,” *Top. Cryptol. - Ct-Rsa 2012*, vol. 7178, pp. 1–18, 2012.

[97] M. Masoomi, M. Masoumi, and M. Ahmadian, “A practical differential power analysis attack against an FPGA implementation of AES cryptosystem,” *2010 Int. Conf. Inf. Soc.*, pp. 308–312, 2010.

[98] F. Khelil, M. Hamdi, S. Guilley, J. L. Danger, and N. Selmane, “Fault Analysis Attack on an FPGA AES Implementation,” in *New Technologies, Mobility and Security, 2008. NTMS ’08.*, 2008, pp. 1–5.

[99] V. Carlier, H. Chabanne, E. Dottax, and H. Pelletier, “Generalizing square attack using side-channels of an aes implementation on an FPGA,” *Proc. - 2005 Int. Conf. F. Program. Log. Appl. FPL*, vol. 2005, pp. 433–437, 2005.

[100] E. De Mulder, P. Buysschaert, S. B. Ors, P. Delmotte, B. Preneel, G. Vandenbosch, and I. Verbauwhede, “Electromagnetic Analysis Attack on an FPGA Implementation of an Elliptic Curve Cryptosystem,” *EUROCON 2005 - Int. Conf. Comput. as a Tool*, vol. 2, 2005.

[101] J. Zhao, J. Han, X. Zeng, L. Li, and Y. Deng, “Differential Power Analysis and Differential Fault Attack Resistant AES Algorithm and its VLSI Implementation,” in *9th International Conference on Solid-State and Integrated-Circuit Technology,*

2008, pp. 2–5.

[102] J. Wu, Y. Shi, and M. Choi, “FPGA-based measurement and evaluation of power analysis attack resistant asynchronous S-Box,” *2011 IEEE Int. Instrum. Meas. Technol. Conf.*, pp. 1–6, 2011.

[103] S. A. Kadir, A. Sasongko, and M. Zulkifli, “Simple power analysis attack against elliptic curve cryptography processor on FPGA implementation,” *Proc. 2011 Int. Conf. Electr. Eng. Informatics, ICEEI 2011*, no. July, pp. 2–5, 2011.

[104] S. Sun, Z. Yan, and J. Zambreno, “Experiments in attacking FPGA-based embedded systems using differential power analysis,” in *2008 IEEE International Conference on Electro/Information Technology*, 2008, pp. 7–12.

[105] Trusted Computing Group, “TPM Main Specification Level 2.” [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-2-TPM-Structures_v1.2_rev116_01032011.pdf.

[106] T. Eisenbarth, T. Güneysu, C. Paar, A.-R. Sadeghi, D. Schellekens, and M. Wolf, “Reconfigurable trusted computing in hardware,” in *Proceedings of the 2007 ACM workshop on Scalable trusted computing - STC ’07*, 2007, p. 15.

[107] B. Glas, A. Klimm, D. Schwab, K. M??ller-Glaser, and J. Becker, “A prototype of trusted platform functionality on reconfigurable hardware for bitstream updates,” *Proc. 19th IEEE/IFIP Int. Symp. Rapid Syst. Prototyp. - Shortening Path from Specif. to Prototype, RSP 2008*, pp. 135–141, 2008.

[108] J. Cucurull and S. Guasch, “Virtual TPM for a secure cloud : fallacy or reality ?,” in *Congresos - RECSI 2014 - Comunicaciones*, 2014, pp. 2–5.

[109] L. Xu, W. Shi, and T. Suh, “PFC: Privacy preserving FPGA cloud - A case study

of MapReduce," in *IEEE International Conference on Cloud Computing, CLOUD*, 2014, pp. 280–287.

[110] T. Kekkonen, T. Kanstrén, and K. Hätönen, "Towards trusted environment in cloud monitoring," *ITNG 2014 - Proc. 11th Int. Conf. Inf. Technol. New Gener.*, pp. 180–185, 2014.

[111] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards Trusted Cloud Computing," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, 2009, vol. 10, no. 2, p. 3.

[112] R. Neisse, D. Holling, and A. Pretschner, "Implementing trust in cloud infrastructures," *Proc. - 11th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. CCGrid 2011*, pp. 524–533, 2011.

[113] B. Bertholon, S. Varrette, and P. Bouvry, "CERTICLOUD: A novel TPM-based approach to ensure cloud IaaS security," in *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011*, 2011, pp. 121–130.

[114] M. Achemlal, S. Gharout, and C. Gaber, "Trusted platform module as an enabler for security in cloud computing," in *2011 Conference on Network and Information Systems Security, SAR-SSI 2011, Proceedings*, 2011.

[115] R. Perez, R. Sailer, and L. van Doorn, "vTPM: virtualizing the trusted platform module," *Usenix.Org*, pp. 305–320, 2005.

[116] S. Nepal, J. Zic, H. Hwang, and D. Moreland, "Trust Extension Device: Providing Mobility and Portability of Trust in Cooperative Information Systems," in *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, 2007, pp. 253–271.

[117] D. Thilakanathan, S. Chen, S. Nepal, R. a. Calvo, D. Liu, and J. Zic, "Secure Multiparty Data Sharing in the Cloud Using Hardware-Based TPM Devices," *2014 IEEE 7th Int. Conf. Cloud Comput.*, pp. 224–231, 2014.

[118] Z. S. Z. Shen and Q. T. Q. Tong, "The security of cloud computing system enabled by trusted computing technology," *Signal Process. Syst. (ICSPS), 2010 2nd Int. Conf.*, vol. 2, pp. 11–15, 2010.

[119] C. Chen, H. Raj, S. Saroiu, I. Nsdi, and A. Wolman, "cTPM : A Cloud TPM for Cross-Device Trusted Applications," *11th USENIX Conf. Networked Syst. Des. Implement.*, vol. 8, pp. 187–201, 2014.

[120] J. Zic and S. Nepal, "Implementing a Portable Trusted Environment," in *Future of Trust in Computing: Proceedings of the First International Conference Future of Trust in Computing 2008: With 58 Illustrations*, 2009, pp. 17–29.

[121] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *HASP'13*, 2013, pp. 1–7.

[122] L. M. Aumasson JP, "SGX Secure Enclaves In Practice: Security And Crypto Review," in *blackhat*, 2016.

[123] C. M. and S. M. M. Schwarz, S. Weiser, D. Gruss, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *arxiv*, 2017.

[124] S. C. and A. S. F. Brasser, U. Muller, A. Dmitrienko, K. Kostiainen, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *arxiv*, 2017.

[125] S. S. and T. M. J. Götzfried, M. Eckert, "Cache Attacks on Intel SGX." Proceedings of the 10th European Workshop on Systems Security, 2017.

[126] and M. P. Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim,

"Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *arXiv*, 2016.

[127] A. Conference, C. Zhao, D. Saifuding, H. Tian, Y. Zhang, and C. Xing, "On the Performance of Intel SGX," in *Web Information Systems and Applications Conference*, 2016.

[128] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," *Proc. Int. Conf. Supercomput.*, pp. 160–171, 2003.

[129] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," *High Perform. Comput. Archit. (HPCA), 2010 IEEE 16th Int. Symp.*, pp. 1–12, 2010.

[130] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," *Proc. 25th USENIX Secur. Symp.*, 2016.

[131] C. W. Fletcher, M. Van Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," *Scalable Trust. Comput. STC '12. Proc. Seventh ACM Work.*, p. 3, 2012.

[132] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "PHANTOM: Practical Oblivious Computation in a Secure Processor," in *CCS'13*, 2013, pp. 311–324.

[133] E. Brickell and J. Li, "Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities," *IEEE Trans. Dependable Secur. Comput.*, vol. 9, no. 3, pp. 345–360, 2012.

[134] John Casey, "Top Five Challenges of Cloud Computing," 2015. [Online].

Available: http://trilogytechnologies.com/top-five-challenges-of-cloud-computing/.

[135] Cloud Standards Customer Council, “Impact of Cloud Computing on Healthcare,” 2012. [Online]. Available: www.cloud-council.org/CSCC-Impact-of-Cloud-Computing-on-Healthcare.pdf.

[136] “Amazon EC2 F1 Instances,” 2017. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/.

[137] B. Guttman and E. Roback, “An Introduction to Computer Security : The NIST Handbook,” *Natl. Inst. Stand. Technol. Technol. Adm. U.S. Dep. Commer. An*, vol. SP800, no. 12, pp. 1–278, 1995.

[138] J. K. B, T. Lin, H. Bannazadeh, and A. Leon-garcia, “Testbeds and Research Infrastructure. Development of Networks and Communities,” in *6th International ICST Conference*, 2012, vol. 44, pp. 3–13.

[139] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, “Enabling FPGAs in Hyperscale Data Centers,” in *IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, 2015, pp. 1078–1086.

[140] Xilinx Inc., “XAPP1084(v1.3): Developing Tamper Resistant Designs with Xilinx Virtex-6 and 7 Series FPGAs,” 2013. [Online]. Available: https://www.xilinx.com/support/documentation/xapp1084_tamp_resist_dsgns.pdf.

[141] S. Goren, O. Ozkurt, A. Yildiz, and H. F. Ugurdag, “FPGA bitstream protection with PUFs, obfuscation, and multi-boot,” in *6th International Workshop on*

Reconfigurable Communication-Centric Systems-on-Chip, ReCoSoC 2011 - Proceedings, 2011.

[142] Xilinx Inc., “Xilinx Partial Reconfiguration User Guide,” 2012. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug702.pdf .

[143] Mark Stamp, “Hash Functions,” in *Information Security Principles and Practice*, 2nd ed., A JOHN WILEY & SONS, INC., PUBLICATION, 2011, p. 608.

[144] S. M. Trimberger and J. J. Moore, “FPGA security: Motivations, features, and applications,” *Proc. IEEE*, vol. 102, no. 8, pp. 1248–1265, 2014.

[145] P. Maistri, S. Tiran, P. Maurine, I. Koren, and R. Leveugle, “Countermeasures against em analysis for a secured FPGA-based AES implementation,” *2013 Int. Conf. Reconfigurable Comput. FPGAs, ReConFig 2013*, 2013.

[146] F. Regazzoni, Y. Wang, and F.-X. Standaert, “FPGA implementations of the AES masked against power analysis attacks,” in *Proceedings of COSADE 2011*, 2011, pp. 56–66.

[147] B. Bahrak and M. R. Aref, “Impossible differential attack on seven-round AES-128,” *IET Inf. Secur.*, vol. 2, no. 2, pp. 28–32, 2008.

[148] K. Tiri and I. Verbauwhede, “A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation,” in *Proceedings - Design, Automation and Test in Europe Conference and Exhibition*, 2004, vol. 1, pp. 246–251.

[149] M. N. Ismail, A. Aborujilah, S. Musa, and A. Shahzad, “New framework to detect and prevent denial of service attack in cloud computing environment,” *Int. J.*

Comput. Sci. Secur., vol. 6, no. 4, pp. 226–237, 2012.

[150] V. Cheval and B. Blanchet, “Proving more observational equivalences with ProVerif,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013, vol. 7796 LNCS, pp. 226–246.

[151] T. Fitfield, “Introduction to OpenStack,” *Linux J.*, vol. 2013, p. 4, 2013.

[152] K. Ward, “Survey: OpenStack Dominates Open Source Cloud,” 2014. [Online]. Available: https://virtualizationreview.com/articles/2014/12/01/openstack-dominates-open-source-cloud.aspx.

[153] C. Research, “Top 15 Open Source Cloud Computing Technologies,” 2014. [Online]. Available: https://www.crisp-research.com/tag/openstack/.

[154] S. Byma and J. Steffan, “FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack,” in *IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 109–116.

[155] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “On the indifferentiability of the sponge construction,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008, vol. 4965 LNCS, pp. 181–197.

[156] M. Litochevski and L. Dongjun, “High throughput and low area AES,” 2012. [Online]. Available: https://opencores.org/project,aes_highthroughput_lowarea.

[157] [45] McQueen, “Basic RSA Encryption Engine.” [Online]. Available: http://opencores.org/project,basicrsa.

[158] D. Suzuki, “How to maximize the potential of FPGA resources for modular

exponentiation,” in *Advances in Cryptology - Cryptographic Hardware and Embedded Systems - CHES 2007*, 2007, pp. 272–288.

[159] T. Good and M. Benaissa, “AES on FPGA from the Fastest to the Smallest,” *Lect. Notes Comput. Sci. Adv. Cryptol. - Cryptogr. Hardw. Embed. Syst. - CHES 2005*, pp. 427–440, 2005.

[160] D. S. Kumar, “Compact Implementation of SHA3-1024 on FPGA,” vol. 3, no. 7, pp. 79–86, 2015.

[161] ITRS, “International Technology Roadmap for Semicon ductors,” 2012. [Online]. Available: http://www.semiconductors.org/clientuploads/Research_Technology/ITRS/2015/0_2015 ITRS 2.0 Executive Report (1).pdf.

[162] Xilinx Inc., “7 Series FPGAs Packaging and Pinout, Product Specification Xilinx.” [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug475_7Series_Pkg_Pinout.pdf.

[163] Akami, “State of the Internet Quarterly Report.” [Online]. Available: https://www.akamai.com.

[164] Dawson, “Chips 2.0.” [Online]. Available: https://github.com/dawsonjon/Chips-Demo.

[165] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, vol. 7. 1973.

[166] S. Brenner, C. Wulf, M. Lorenz, N. Weichbrodt, D. Goltzsche, C. Fetzer, P. Pietzuch, and R. Kapitza, “SecureKeeper: Confidential ZooKeeper using Intel SGX,” *Middlew. 2016*, 2016.

[167] J. Rivera and R. Van der Muelen, “Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020,” *Gartner*, 2013. .

[168] R. van Kranenburg and A. Bassi, “IoT Challenges,” *Commun. Mob. Comput.*, vol. 1, no. 1, p. 9, 2012.

[169] Amazon, “AWS IoT,” *Amazon*, 2016. [Online]. Available: https://aws.amazon.com/iot/.

[170] TechTarget, “Reference Architecture,” 2012. [Online]. Available: http://internetofthingsagenda.techtarget.com/definition/reference-architecture.

[171] M. J. Yuan, “IBM Watson,” *Ibm*, 2011. .

[172] Microsoft, “Azure and Internet of Things,” 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-suite/iot-suite-what-is-azure-iot.

[173] A. M. Amitranjan Gantait, Joy Patra, “Securing IoT devices and gateways.” IBM, 2016.

[174] Google, “Overview of Internet of Things.” [Online]. Available: https://cloud.google.com/solutions/iot-overview.

[175] “Intel IoT Platform.” [Online]. Available: https://www.intel.com/content/www/us/en/internet-of-things/infographics/iot-platform-infographic.html. [Accessed: 28-Jun-2017].

[176] H. Ning, H. Liu, and L. T. Yang, “Aggregated-proof based hierarchical authentication scheme for the internet of things,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 657–667, 2015.

[177] N. Ye, Y. Zhu, R. C. Wang, R. Malekian, and Q. M. Lin, “An efficient authentication and access control scheme for perception layer of internet of

things," *Appl. Math. Inf. Sci.*, vol. 8, no. 4, pp. 1617–1624, 2014.

[178] R. Hummen, J. H. Ziegeldorf, H. Shafagh, S. Raza, and K. Wehrle, "Towards viable certificate-based authentication for the internet of things," *Proc. 2nd ACM Work. Hot Top. Wirel. Netw. Secur. Priv. - HotWiSec '13*, p. 37, 2013.

[179] F. Li and P. Xiong, "Practical secure communication for integrating wireless sensor networks into the internet of things," *IEEE Sens. J.*, vol. 13, no. 10, pp. 3677–3684, 2013.

[180] I. E. Bagci, S. Raza, T. Chung, U. Roedig, and T. Voigt, "Combined secure storage and communication for the Internet of Things," in *2013 IEEE International Conference on Sensing, Communications and Networking, SECON 2013*, 2013, pp. 523–531.

[181] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt, "Lithe: Lightweight secure CoAP for the internet of things," *IEEE Sens. J.*, vol. 13, no. 10, pp. 3711–3720, 2013.

[182] H. Zhang and T. Zhang, "Short Paper: 'A peer to peer security protocol for the internet of things': Secure communication for the sensiblethings platform," in *2015 18th International Conference on Intelligence in Next Generation Networks, ICIN 2015*, 2015, pp. 154–156.

[183] J. Qian, H. Xu, and P. Li, "A novel secure architecture for the internet of things," in *Proceedings - 2016 International Conference on Intelligent Networking and Collaborative Systems, IEEE INCoS 2016*, 2016, pp. 398–401.

[184] D. Singh, G. Tripathi, and A. Jara, "Secure layers based architecture for Internet of Things," *IEEE World Forum Internet Things, WF-IoT 2015 - Proc.*, pp. 321–326,

2015.

[185] S. Sicari, A. Rizzardi, D. Miorandi, C. Cappiello, and A. Coen-Porisini, “A secure and quality-aware prototypical architecture for the Internet of Things,” *Inf. Syst.*, vol. 58, pp. 43–55, 2016.

[186] T. Bhattasali, R. Chaki, and N. Chaki, “Secure and trusted cloud of things,” in *2013 Annual IEEE India Conference, INDICON 2013*, 2013.

[187] B. S. Xavier Allamigeon, Vincent Cheval, “ProVerif: Cryptographic protocol verifier in the formal model,” 2014. [Online]. Available: http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[188] A. P. Fournaris, “Fault and simple power attack resistant RSA using Montgomery modular multiplication,” in *ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems*, 2010, pp. 1875–1878.

[189] D. N. Amanor, C. Paar, J. Pelzl, V. Bunimov, and M. Schimmler, “Efficient hardware architectures for modular multiplication on FPGAs,” in *Proceedings - 2005 International Conference on Field Programmable Logic and Applications, FPL*, 2005, vol. 2005, pp. 539–542.

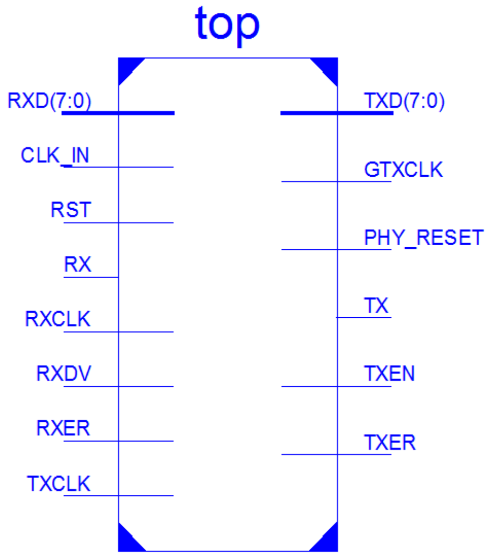
[190] M. Attig and G. Brebner, “400 Gb/s programmable packet parsing on a single FPGA,” in *Proceedings - 2011 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2011*, 2011, pp. 12–23.

[191] cloudadmin, “Cloud Computing Architecture,” 2015. [Online]. Available: http://cloudcomputingnet.com/cloud-computing-architecture/.

APPENDICES

APPENDIX A: FPGA Implementation and Simulation

a. Client Data Protection Scheme Components



RXD	input	8-bit bus for receiving data from the Ethernet chip
RX	input	Unused signal
RXDV	input	Valid signal for the received data
RXER	input	Error signal for the received data
TXD	output	8-bit bus for transmitting data to the Ethernet chip
TX	output	Unused signal
TXEN	output	Enable signal for data transmission
TXER	output	Error signal for the Transmitted data

Figure A 1: Top module inputs/outputs

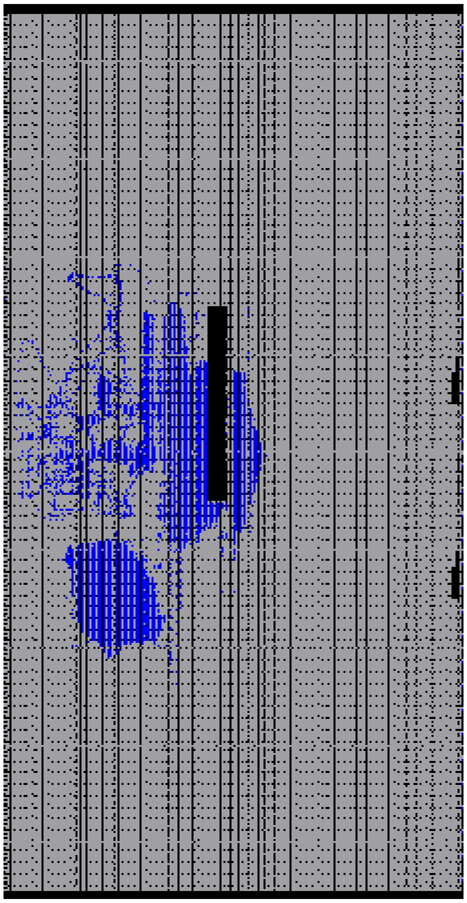
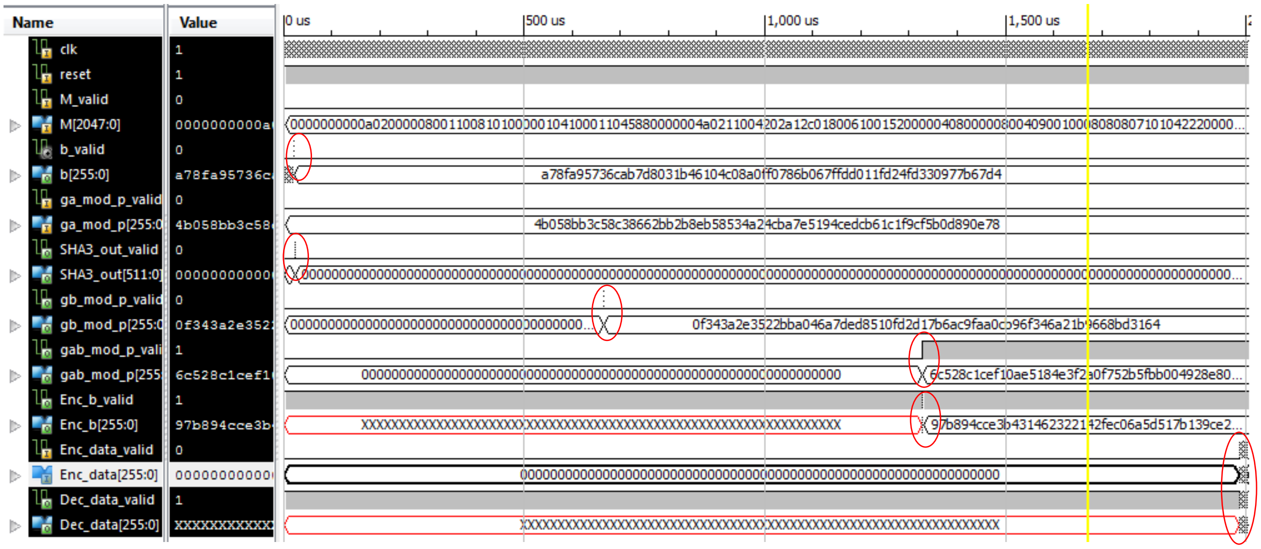


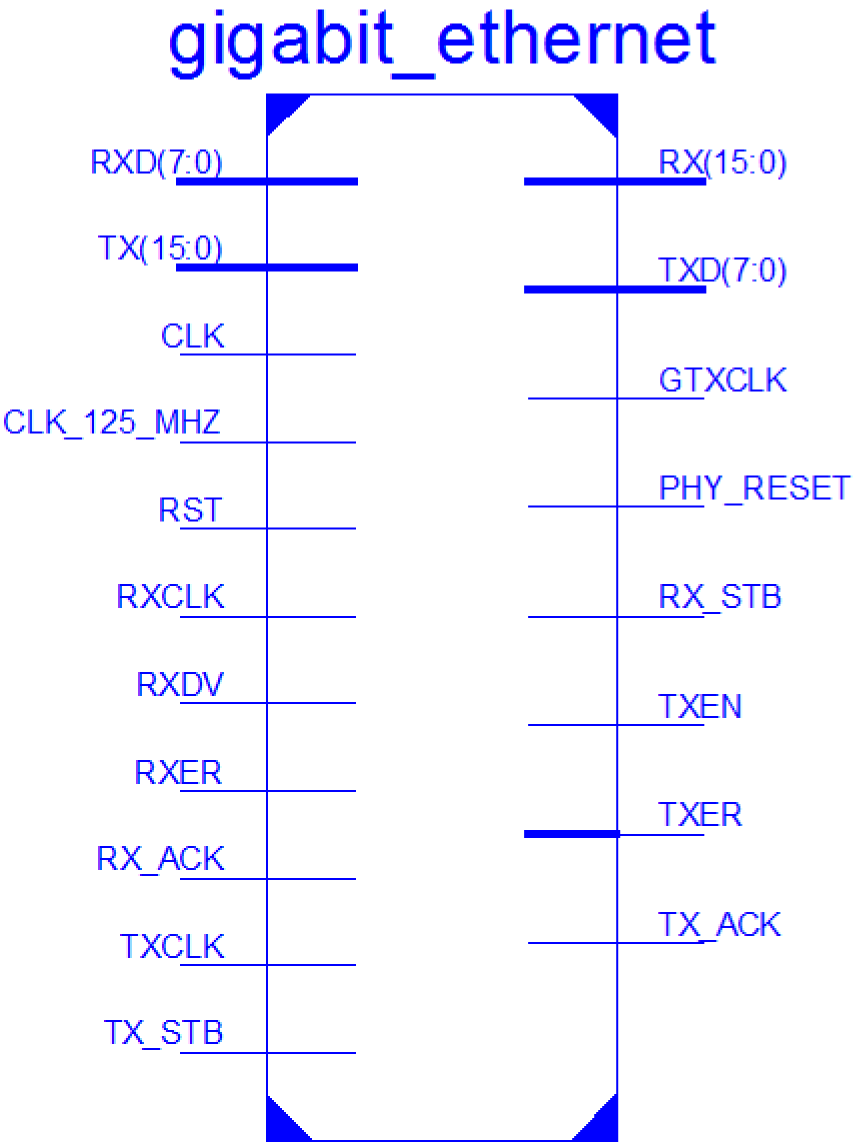
Figure A 2: Implementation of the design placed in Xilinx Virtex 6 device.



Name	Value
clk	1
reset	1
M_valid	0
M[2047:0]	0000000000a
b_valid	0
b[255:0]	a78fa95736c
ga_mod_p_valid	0
ga_mod_p[255:0	4b058bb3c58
SHA3_out_valid	0
SHA3_out[511:0]	00000000000
gb_mod_p_valid	0
gb_mod_p[255:0	0f343a2e352
gab_mod_p_vali	1
gab_mod_p[255	6c528c1cef1
Enc_b_valid	1
Enc_b[255:0]	97b894cce3b
Enc_data_valid	0
Enc_data[255:0]	00000000000
Dec_data_valid	1
Dec_data[255:0]	XXXXXXXXXXX

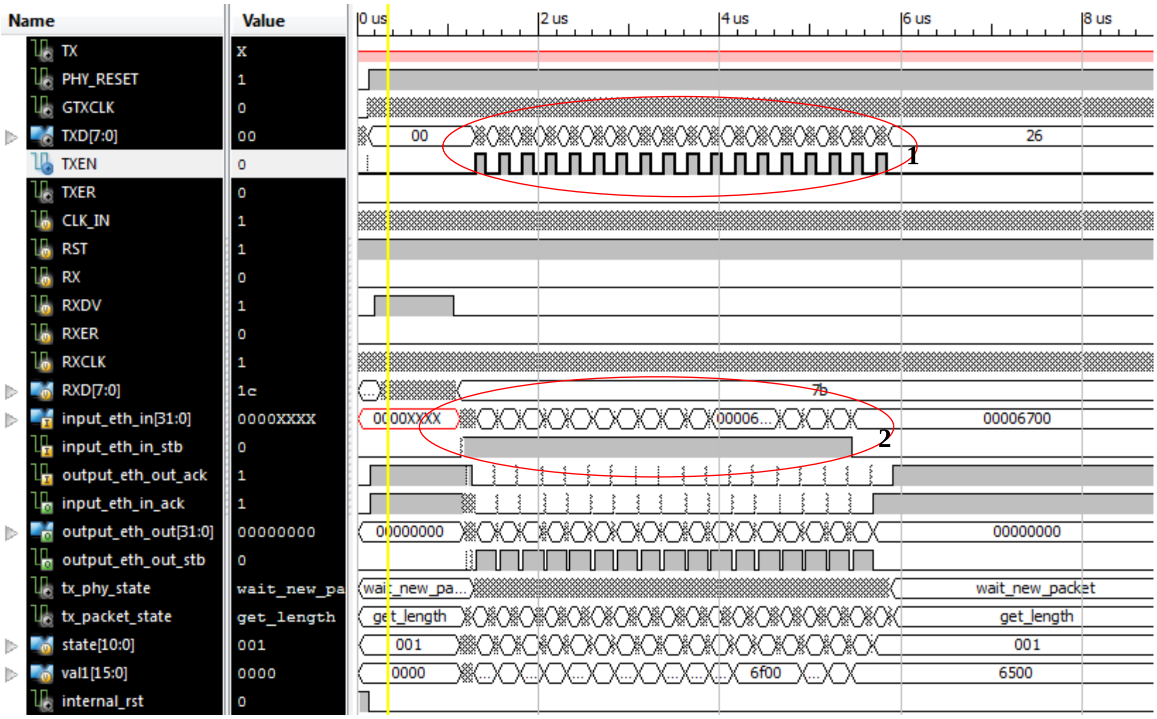
0 us
500 us
1,000 us
1,500 us
0000000000a02000008001100810100000104100011045880000004a0211004202a12c01800610015200000408000008004090010008080807101042220000...
a78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4
4b058bb3c58c38662bb2b8eb58534a24cba7e5194cedcb61c1f9cf5b0d890e78
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000...
000000000000000000000000000000000000000...
0f343a2e3522bba046a7ded8510fd2d17b6ac9faa0cb96f346a21b9668bd3164
00000000000000000000000000000000000000000000000000000000000
6c528c1cef10ae5184e3f2a0f752b5fbb004928e80...
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
97b894cce3b431462322142fec06a5d517b139ce2...
0000000000000000000000000000000000000000000000000000000000000000
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Figure A 3: Simulation of the top module of the design



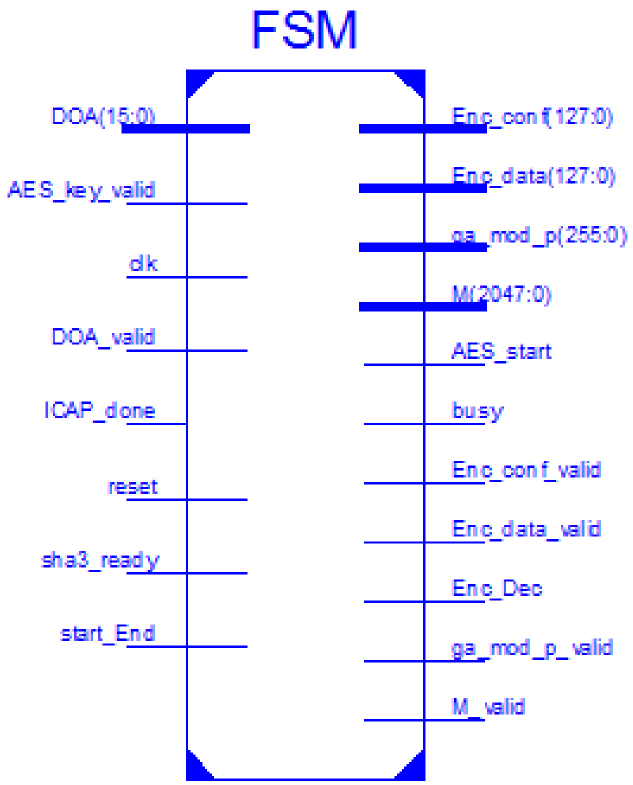
TX_STB	input	Transmit strobe
RX_STB	output	Receive strobe

Figure A 4: Ethernet controller block



Name	Value
TX	X
PHY_RESET	1
GTXCLK	0
TXD[7:0]	00
TXEN	0
TXER	0
CLK_IN	1
RST	1
RX	0
RXDV	1
RXER	0
RXCLK	1
RXD[7:0]	1c
input_eth_in[31:0]	0000XXXX
input_eth_in_stb	0
output_eth_out_ack	1
input_eth_in_ack	1
output_eth_out[31:0]	00000000
output_eth_out_stb	0
tx_phy_state	wait_new_pa
tx_packet_state	get_length
state[10:0]	001
val1[15:0]	0000
internal_rst	0

Figure A 5: Ethernet controller simulation (1) transmitting a packet (2) receiving a packet.



DOA	input	16-bit bus for data received from the Ethernet controller
Start_End	input	Signal to start/end the session
AES_key_valid	input	Signal to indicate that the AES can encrypt/decrypt the data
busy	output	Signal to indicate that the FSM is not able to receive new data from the Ethernet controller
AES_start	output	Signal is used for the AES key expansion which is performed at the beginning of the session.
Enc_Dec	output	Signal for the AES to encrypt or decrypt the data

Figure A 6: Main FSM module

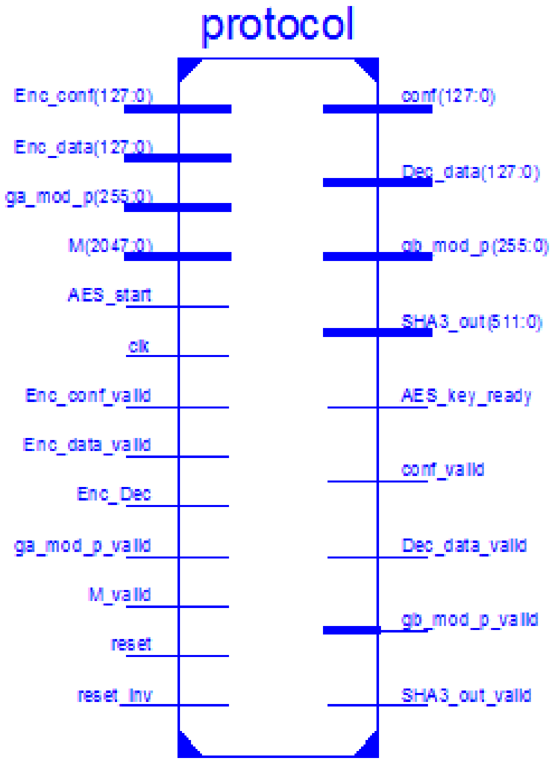


Figure A 7: Protocol block and inputs/outputs

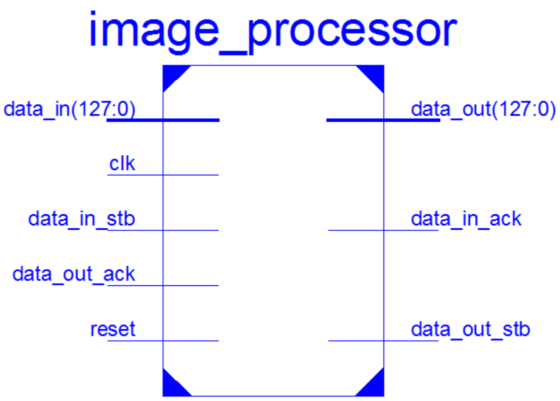
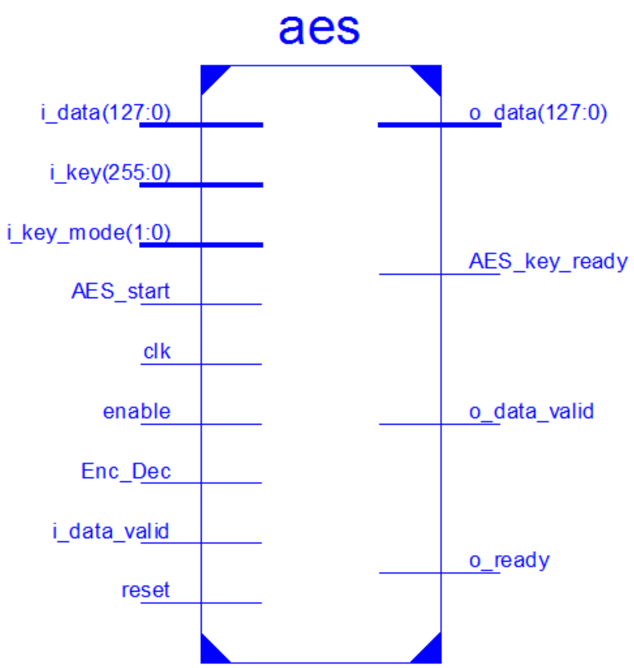
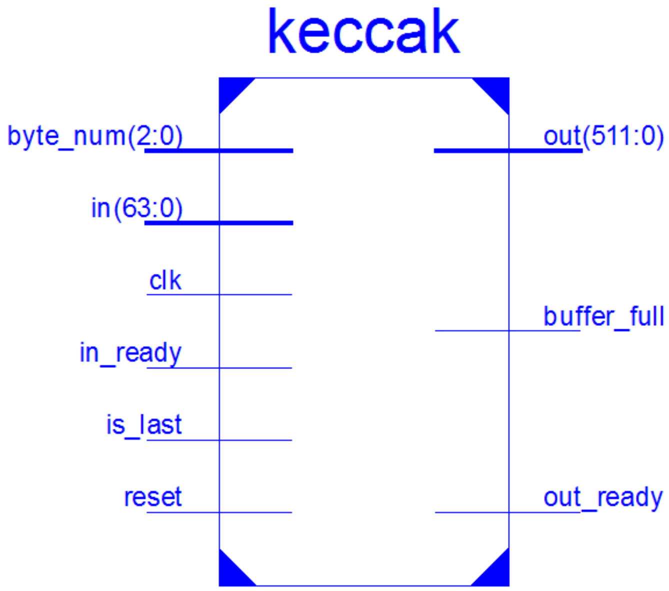


Figure A 8: Image processor module



i_key_mode	input	2-bit bus for specifying the key length (0 = 128; 1 = 192; 2 = 256)
o_data_valid	output	data output valid
o_ready	output	indicates AES is ready for new input data at the next clock cycle

Figure A 9: AES module



byte_num	input	Number of bytes per block
is last	input	Signal to indicate the last block
buffer_full	output	Unused signal. For debugging purposes

Figure A 10: SHA3 module

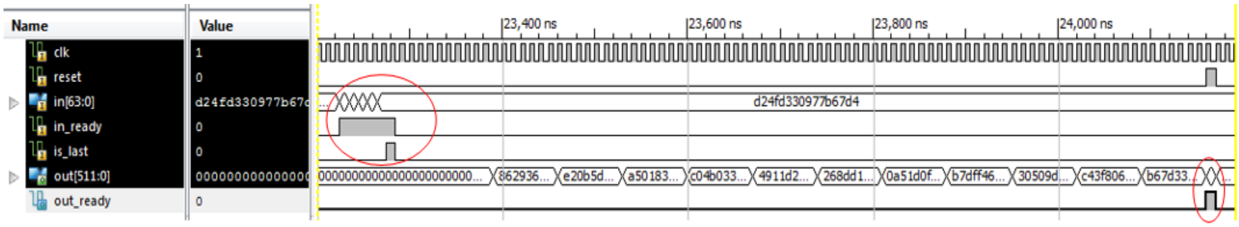


Figure A 11: The operations of SHA3

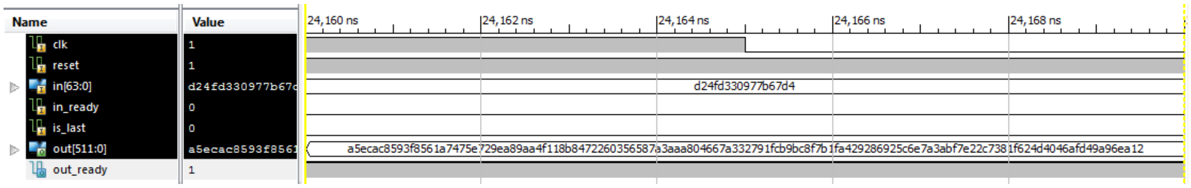
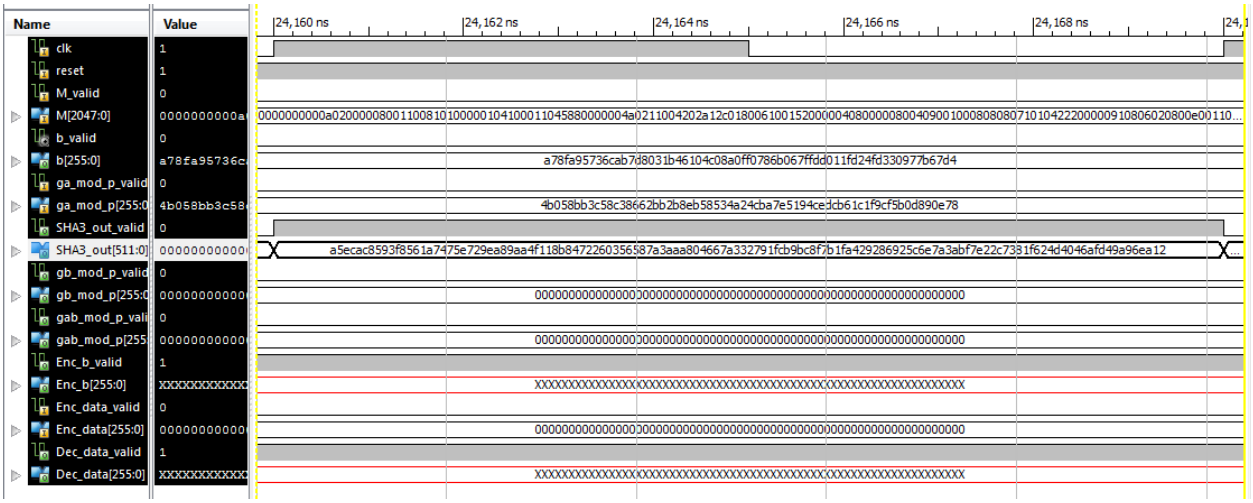


Figure A 12: The output of SHA3 when out_ready goes high



Name	Value	
clk	1	
reset	1	
M_valid	0	
M[2047:0]	0000000000a	0000000000a02000008001100810100000104100011045880000004a0211004202a12c018006100152000004080000080040900100080808071010422200000910806020800e00110...
b_valid	0	
b[255:0]	a78fa95736c	a78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4
ga_mod_p_valid	0	
ga_mod_p[255:0	4b058bb3c58	4b058bb3c58c38662bb2b8eb58534a24cba7e5194cedcb61c1f9cf5b0d890e78
SHA3_out_valid	0	
SHA3_out[511:0]	00000000000	a5ecac8593f8561a7475e729ea89aa4f118b8472260356587a3aaa804667a332791fcb9bc8f7b1fa429286925c6e7a3abf7e22c7381f624d4046afd49a96ea12
gb_mod_p_valid	0	
gb_mod_p[255:0	00000000000	0000000000000000000000000000000000000000000000000000000000000000
gab_mod_p_vali	0	
gab_mod_p[255	00000000000	0000000000000000000000000000000000000000000000000000000000000000
Enc_b_valid	1	
Enc_b[255:0]	XXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Enc_data_valid	0	
Enc_data[255:0]	00000000000	0000000000000000000000000000000000000000000000000000000000000000
Dec_data_valid	1	
Dec_data[255:0]	XXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Figure A 13: The result of SHA3

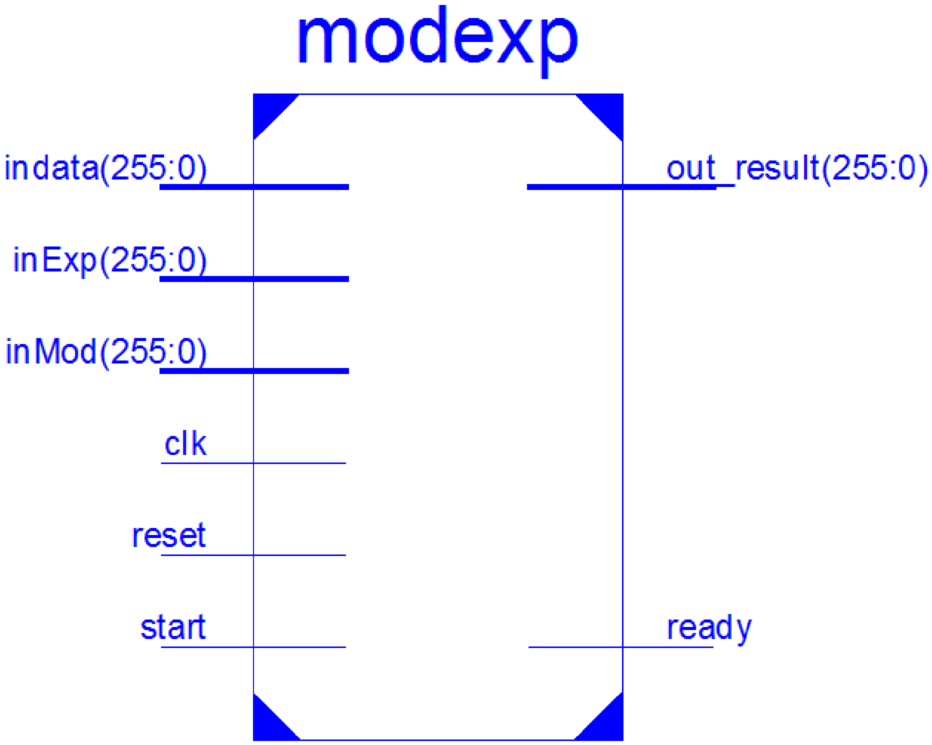


Figure A 14: Modular exponentiation module

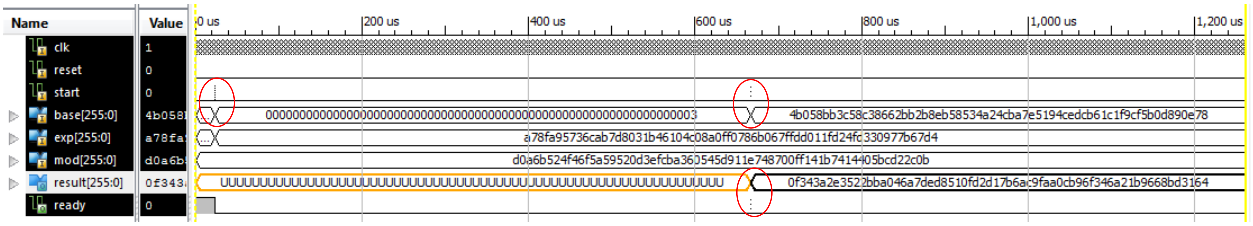

Name
Value
0 us
200 us
400 us
600 us
800 us
1,000 us
1,200 us
clk 1
reset 0
start 0
base[255:0] 4b058
exp[255:0] a78fa
mod[255:0] d0a6b
result[255:0] 0f343
ready 0
0000000000000000000000000000000000000000000000000000000000000003
4b058bb3c58c38662bb2b8eb58534a24cba7e5194cedcb61c1f9cf5b0d890e78
a78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4
d0a6b524f46f5a59520d3efcba360545d911e748700ff141b7414405bcd22c0b
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
0f343a2e3522bba046a7ded8510fd2d17b6ac9faa0cb96f346a21b9668bd3164

Figure A 15: The operations of modexp, producing 3^b mod p (b is exp and p is mod in the figure)

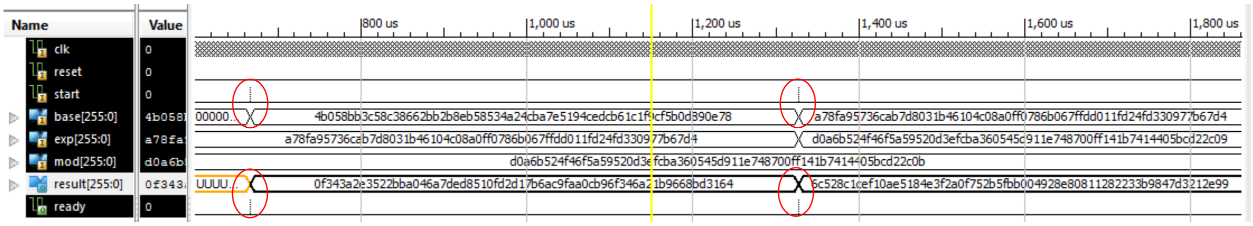

Name
Value
800 us
1,000 us
1,200 us
1,400 us
1,600 us
1,800 us
clk 0
reset 0
start 0
base[255:0] 4b058
exp[255:0] a78fa
mod[255:0] d0a6b
result[255:0] 0f343
ready 0
00000...
4b058bb3c58c38662bb2b8eb58534a24cba7e5194cedcb61c1f9cf5b0d890e78
a78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4
a78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4
d0a6b524f46f5a59520d3efcba360545d911e748700ff141b7414405bcd22c09
d0a6b524f46f5a59520d3efcba360545d911e748700ff141b7414405bcd22c0b
UUUU...
0f343a2e3522bba046a7ded8510fd2d17b6ac9faa0cb96f346a21b9668bd3164
6c528c1cef10ae5184e3f2a0f752b5fbb004928e80811282233b9847d3212e99

Figure A 16: The operations of modexp, producing 3^{ab} mod p

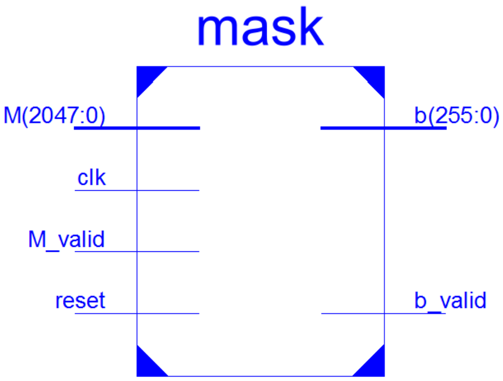


Figure A 17: Masking circuitry module

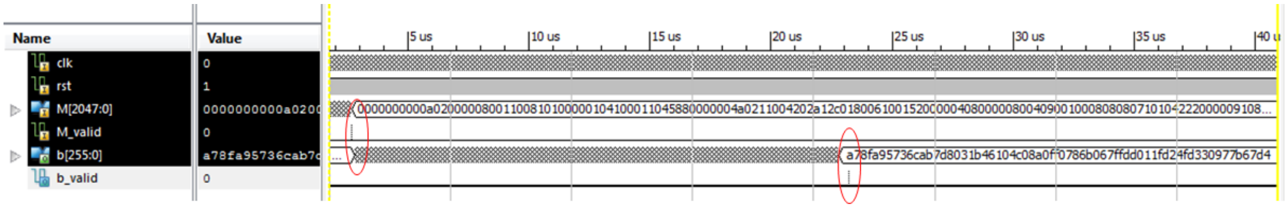

Name
Value
5 us
10 us
15 us
20 us
25 us
30 us
35 us
40 u
clk 0
rst 1
M[2047:0] 0000000000a020
M_valid 0
b[255:0] a78fa95736cab7
b_valid 0
0000000000a0200000800110081010000010410001104588000004a0211004202a12c018006100152000004080000008004090010008080807101042220000009108...
...
a78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4

Figure A 18: The operations of the masking circuitry

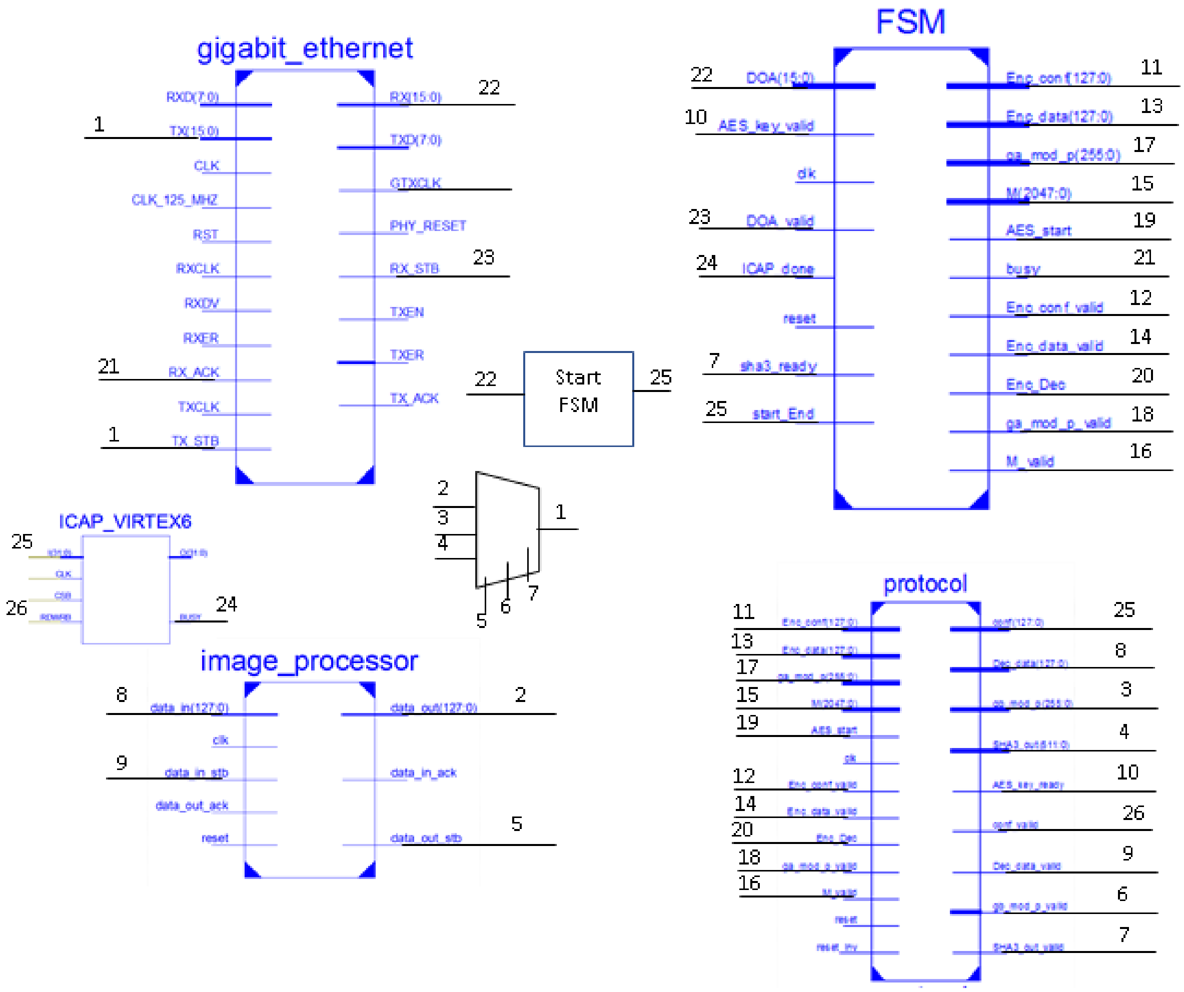


Figure A 19: Simplified view of the top components

b. IoT Scheme Implementation Related Components:

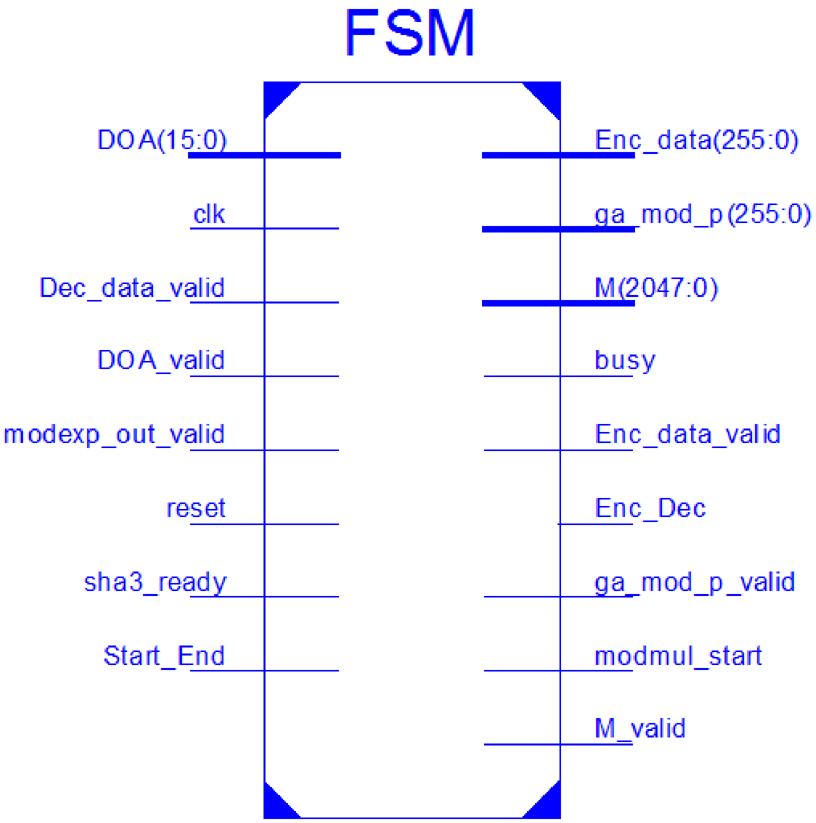


Figure A 20: Main FSM module

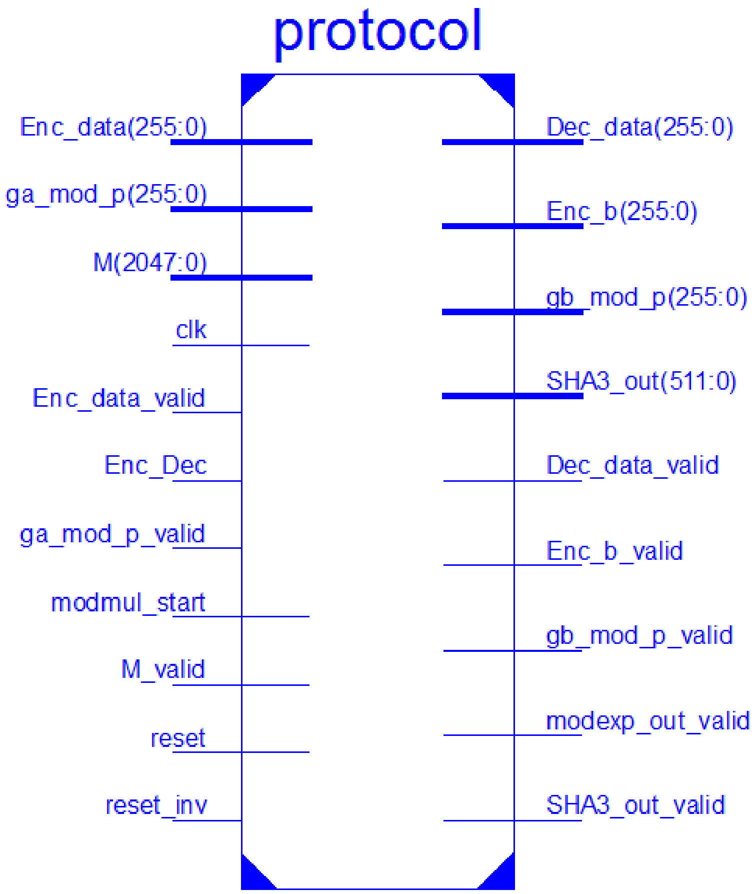


Figure A 21: Protocol module

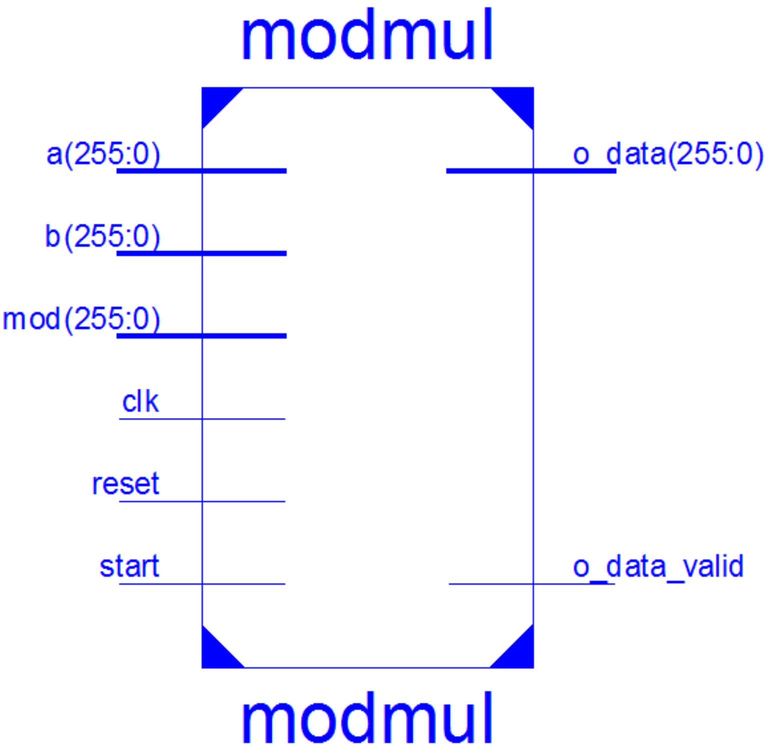
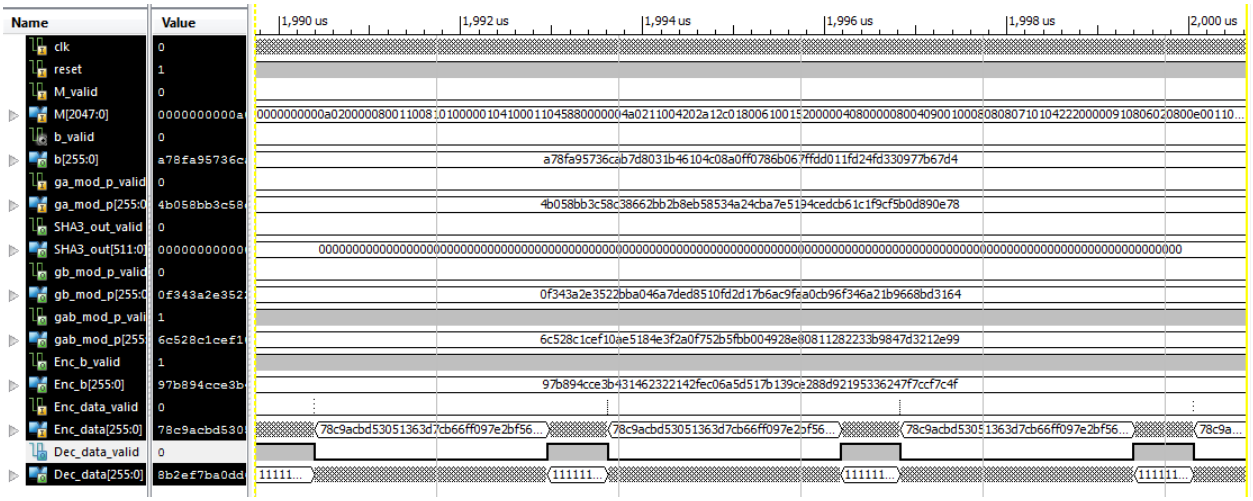


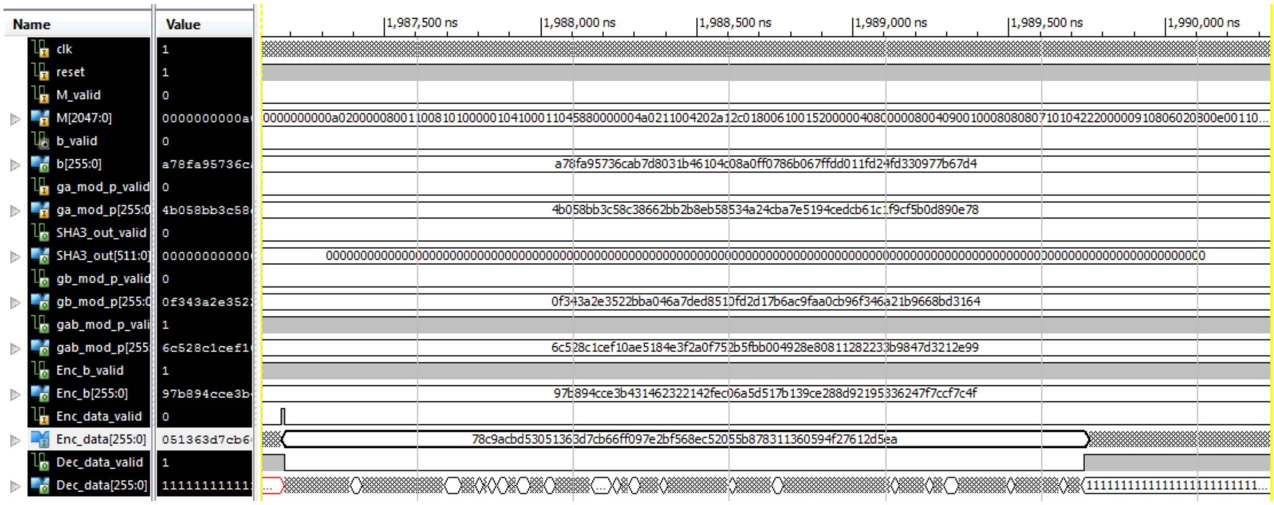
Figure A 22: Modular multiplication module



Name	Value
clk	0
reset	1
M_valid	0
M[2047:0]	0000000000a
b_valid	0
b[255:0]	a78fa95736c
ga_mod_p_valid	0
ga_mod_p[255:0	4b058bb3c58
SHA3_out_valid	0
SHA3_out[511:0]	000000000000
gb_mod_p_valid	0
gb_mod_p[255:0	0f343a2e352
gab_mod_p_vali	1
gab_mod_p[255	6c528c1cef1
Enc_b_valid	1
Enc_b[255:0]	97b894cce3b
Enc_data_valid	0
Enc_data[255:0]	78c9acbd530
Dec_data_valid	0
Dec_data[255:0]	8b2ef7ba0dd

1,990 us
1,992 us
1,994 us
1,996 us
1,998 us
2,000 us
0000000000a02000008001100810100000104100011045880000004a0211004202a12c018006100152000004080000080040900100080808071010422200000910806020800e00110...
a78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4
4b058bb3c58c38662bb2b8eb58534a24cba7e5194cedcb61c1f9cf5b0d890e78
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0f343a2e3522bba046a7ded8510fd2d17b6ac9faa0cb96f346a21b9668bd3164
6c528c1cef10ae5184e3f2a0f752b5fbb004928e80811282233b9847d3212e99
97b894cce3b431462322142fec06a5d517b139ce288d92195336247f7ccf7c4f
78c9acbd53051363d7cb66ff097e2bf56...
78c9acbd53051363d7cb66ff097e2bf56...
78c9acbd53051363d7cb66ff097e2bf56...
78c9a...
11111...
111111...
111111...
111111...

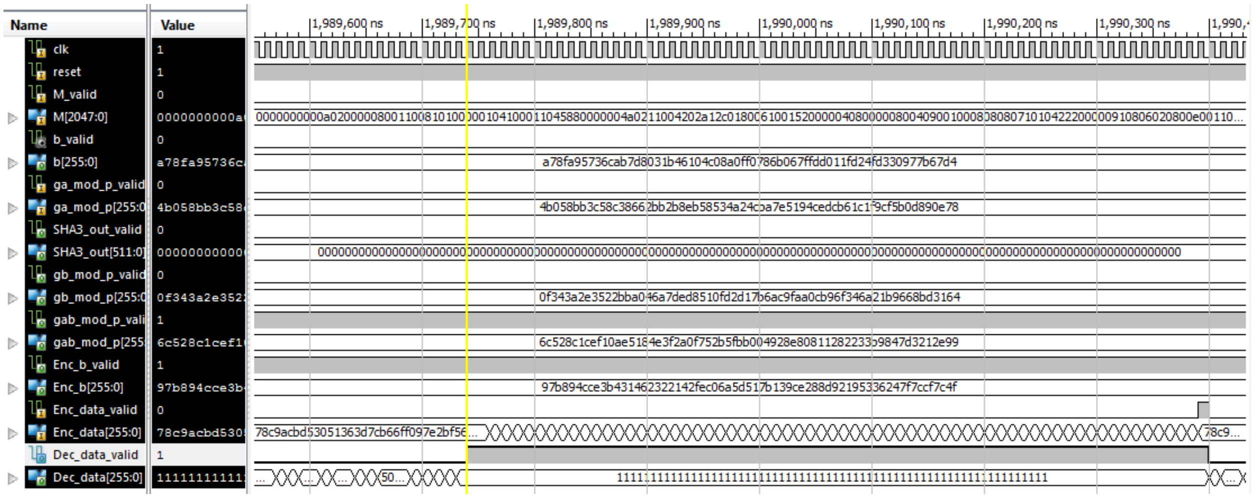
Figure A 23: Encrypting and decrypting data; all other values are also shown such as M, b, etc.



Name	Value
clk	1
reset	1
M_valid	0
M[2047:0]	0000000000a
b_valid	0
b[255:0]	a78fa95736c
ga_mod_p_valid	0
ga_mod_p[255:0	4b058bb3c58
SHA3_out_valid	0
SHA3_out[511:0]	00000000000
gb_mod_p_valid	0
gb_mod_p[255:0	0f343a2e352
gab_mod_p_vali	1
gab_mod_p[255	6c528c1cef1
Enc_b_valid	1
Enc_b[255:0]	97b894cce3b
Enc_data_valid	0
Enc_data[255:0]	051363d7cb6
Dec_data_valid	1
Dec_data[255:0]	11111111111

1,987,500 ns
1,988,000 ns
1,988,500 ns
1,989,000 ns
1,989,500 ns
1,990,000 ns
0000000000a0200000800110081010000010410001104588000000004a0211004202a12c0180061001520000040800000800409001000808080710104222000009108060208 00e00110...
a78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4
4b058bb3c58c38662bb2b8eb58534a24cba7e5194cedcb61c1f9cf5b0d890e78
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0f343a2e3522bba046a7ded8510fd2d17b6ac9faa0cb96f346a21b9668bd3164
6c528c1cef10ae5184e3f2a0f752b5fbb004928e80811282233b9847d3212e99
97b894cce3b431462322142fec06a5d517b139ce288d92195336247f7ccf7c4f
78c9acbd53051363d7cb66ff097e2bf568ec52055b878311360594f27612d5ea
...
...
1111111111111111111111111...

Figure A 24: The value of the encrypted data zoomed



Name	Value
clk	1
reset	1
M_valid	0
M[2047:0]	0000000000a
b_valid	0
b[255:0]	a78fa95736c
ga_mod_p_valid	0
ga_mod_p[255:0	4b058bb3c58
SHA3_out_valid	0
SHA3_out[511:0]	00000000000
gb_mod_p_valid	0
gb_mod_p[255:0	0f343a2e352
gab_mod_p_vali	1
gab_mod_p[255	6c528c1cef1
Enc_b_valid	1
Enc_b[255:0]	97b894cce3b
Enc_data_valid	0
Enc_data[255:0]	78c9acbd530
Dec_data_valid	1
Dec_data[255:0]	11111111111

1,989,600 ns
1,989,700 ns
1,989,800 ns
1,989,900 ns
1,990,000 ns
1,990,100 ns
1,990,200 ns
1,990,300 ns
1,990,
0000000000a02000008001100810100000104100011045880000004a0211004202a12c01800610015200000408000008004090010008080807101042220000091080602080 0e00110...
a78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4
4b058bb3c58c38662bb2b8eb58534a24cba7e5194cedcb61c1f9cf5b0d890e78
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0f343a2e3522bba046a7ded8510fd2d17b6ac9faa0cb96f346a21b9668bd3164
6c528c1cef10ae5184e3f2a0f752b5fbb004928e80811282233b9847d3212e99
97b894cce3b431462322142fec06a5d517b139ce288d92195336247f7ccf7c4f
78c9acbd53051363d7cb66ff097e2bf56...
78c9...
...
...
...
50...
11111111111111111111111111111111111111111111111111111111111111111
...

Figure A 25: The value of decrypted data zoomed

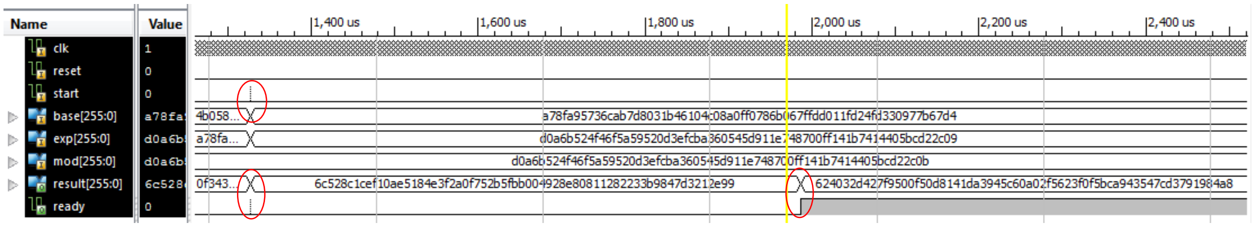

Name
Value
1,400 us
1,600 us
1,800 us
2,000 us
2,200 us
2,400 us
clk 1
reset 0
start 0
base[255:0] a78fa
4b058...
a78fa95736cab7d8031b46104c08a0ff0786b067ffdd011fd24fd330977b67d4
exp[255:0] d0a6b
a78fa...
d0a6b524f46f5a59520d3efcba360545d911e748700ff141b7414405bcd22c09
mod[255:0] d0a6b
d0a6b524f46f5a59520d3efcba360545d911e748700ff141b7414405bcd22c0b
result[255:0] 6c528
0f343...
6c528c1cef10ae5184e3f2a0f752b5fbb004928e80811282233b9847d3212e99
624032d427f9500f50d8141da3945c60a02f5623f0f5bca943547cd3791984a8
ready 0

Figure A 26: The operations of modexp, producing the multiplicative modular inverse of b

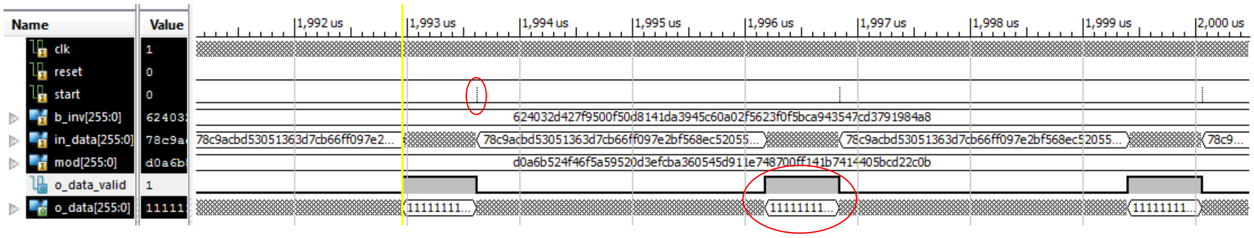

Name
Value
1,992 us
1,993 us
1,994 us
1,995 us
1,996 us
1,997 us
1,998 us
1,999 us
2,000 us
clk 1
reset 0
start 0
b_inv[255:0] 62403
624032d427f9500f50d8141da3945c60a02f5623f0f5bca943547cd3791984a8
in_data[255:0] 78c9a
78c9acbd53051363d7cb66ff097e2...
78c9acbd53051363d7cb66ff097e2bf568ec52055...
78c9acbd53051363d7cb66ff097e2bf568ec52055...
78c9...
mod[255:0] d0a6b
d0a6b524f46f5a59520d3efcba360545d911e748700ff141b7414405bcd22c0b
o_data_valid 1
o_data[255:0] 11111
11111111...
11111111...
11111111...

Figure A 27: The operations of the modular multiplication

APPENDIX B: ProVerif Code for the client sensitive data protection protocol

```
(* Diffie-Hellman without signatures resists active attacks
         A -> B : e^n0
        B -> A : e^n1
         A and B compute the key as k = (e^n0)^n1 = (e^n1)^n0
        A -> B : {s}k
*)
free c.(*a channel used to send/receive messages between the parties *)
free c1. (*a channel used to send/receive messages between the TA and the client/FPGA *)
private free s. (*a message to be send securely upon executing the protocol *)
(* active adversary *)
param attacker = active. (*Active means that the attacker can intercept messages send ,receive or modify messages *)
(* Shared key cryptography *)
fun enc/2. (*encryption function with 2 inputs *)
reduc dec(enc(x,y),y) = x. (*the corresponding decryption*)
fun hash/1. (* the hash function with 1 input *)
(* Diffie-Hellman functions *)
fun f/2. (*a function used to represent $g^{ab}$=$g^{ba}$ *)
fun g/1. (*the exponent ion function *)
```

```
equation f(x,g(y)) = f(y,g(x)). (*the corresponding equation of the function f *)

(* Test whether message s is secret *)

query attacker:s.

(* The TA process *)

let TA = new n00; new b; new n1;

(*using channel c1 to share a key with the client and the FPGA*)

      (out(c1,g(n00)) | in(c1,x11); let k = f(n00,x11) in out(c1,
enc(g(n1),k));out(c1,enc(hash(b),k))).

(*The client process *)

let client = new n0; new n11;

(*sharing a key with the TA *)

      (out(c1,g(n11)) | in(c1,xx);

                 let k = f(n11,xx) in

                 in (c1,m);

(*receiving gb and hash(b) from the TA *)

                 let gb_TA = dec(m,k) in

       in (c1,m1);

                 let hash_b = dec(m,k) in

(*receiving gb and hash(b) from the FPGA *)

       out(c,g(n0)) ; in(c,gb_FPGA)| in(c,hash_FPGA);

(*authenticating the FPGA *)

       if gb_TA=gb_FPGA then

      (
        if hash_b=hash_FPGA then
(*if authentication done, send the message s *)
        let k1 = f(n0,gb_TA) in out(c, enc(s,k1))
        )
        else
       (
        0)
       ).
(*The FPGA process *)
let FPGA =  new n01;
(*sharing a key with the TA *)
       (out(c1,g(n01)) | in(c1,yy);
                 let k = f(n01,yy) in
                   in (c1,m1);
(*receiving gb and hash(b) from the TA *)
                   let gb = dec(m1,k) in
        in (c1,m2);
                   let hash_b1 = dec(m2,k) in
        in(c,x0);
                   let k1 = f(gb,x0) in
(*sending gb and hash(b) to the client*)
        out(c,gb);
        out(c,hash_b1);
```

```
                in (c,m3);
(*receiving the message s  from the client *)
                let s3 = dec(m3,k1) in 0).


process TA | client| FPGA
```

APPENDIX C: Proverif code for the IoT sensitive data protection protocol

```
1.  (* Diffie-Hellman representation
2.  A -> B : e^n0
            B -> A : e^n1
            A and B compute the key as k = (e^n0)^n1 = (e^n1)^n0
3.  A -> B : {s}k *)
4.  free c.(*a channel used to send/receive messages between the parties *)
5.  free c1. (*a channel used to send/receive messages between the TA and the
    IoT_device/FPGA *)
6.  private free s. (*a message to be send securely upon executing the protocol *)
7.  (* active adversary *)
8.  param attacker = active. (*Active means that the attacker can intercept messages send
    ,receive or modify messages *)
9.  (* Shared key cryptography *)
10. fun enc/2. (*encryption function with 2 inputs *)
11. reduc dec(enc(x,y),y) = x. (*the corresponding decryption*)
12. fun hash/1. (* the hash function with 1 input *)
13. (* Diffie-Hellman functions *)
14. fun f/2. (*a function used to represent $g^{ab}=g^{ba}$ *)
15. fun g/1. (*the exponent ion function *)
16. equation f(x,g(y)) = f(y,g(x)). (*the corresponding equation of the function f *)
17. (* Test whether message s is secret *)
18. query attacker:s.
19. (* The TA process *)
20. let TA = new n00; new b;
21. (*using channel c1 to share a key with the IoT_device and the FPGA*)
22. (out(c1,g(n00))  | in(c1,x11); let k = f(n00,x11) in out(c1,
    enc(g(n1),k));out(c1,enc(hash(b),k))).
23. (*The IoT_device process *)
24. let IoT_device = new n0; new n11;
25. (*sharing a key with the TA *)
26. (*receiving gb and hash(b) from the TA *)
            let gb_TA = in (c1,m);
            let hash_b = in (c1,h) in
27. (*receiving gb and hash(b) from the FPGA *)
                    out(c,g(n0)) ; in(c,gb_FPGA)| in(c,hash_FPGA);
28. (*authenticating the FPGA *)
                    if gb_TA=gb_FPGA then
                    (
                    if hash_b=hash_FPGA then
29. (*if authentication done, send the message s *)
```

```
                    let k1 = f(n0,gb_TA) in out(c, enc(s,k1))
                    )else(0)).
30. (*The FPGA process *)
31. let FPGA =  new n01;
32. (*receiving gb and hash(b) from the TA and receiving ga from the client *)
          let gb = in (c1,m);
          let hash_b1 = in (c1,h); in
                    in(c,x0);
          let k1 = f(gb,x0) in
33. (*sending gb and hash(b) to the IoT_device*)
                    out(c,gb);
                    out(c,hash_b1);
          in (c,m3);
34. (*receiving the message s  from the IoT_device *)
          let s3 = dec(m3,k1) in 0).
35. process TA | IoT_device | FPGA
```

APPENDIX D: Python Scripts

The TA:

```
import numpy as np
import random
import math
from keccak import *
g = 3
p = pow(2,256)
import socket
# create a socket object
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# get local machine name
IOTdevice_IP = ""
IOTdevice_port = 9998
s.bind((IOTdevice_IP, IOTdevice_port))
s.listen(1)
conn, addr = s.accept()
print 'Connection address:', addr
# Receive FPGA ID
FPGA_ID = conn.recv(1024)
N = 2048
```

```
K = 256 #K ones

M = np.array([0] * (N-K) + [1] * K)

zeros = ['0']*(N-K)

ones = ['1']*K

M = zeros+ones

random.shuffle(M)


with open('RN.txt') as f:

    arr01 = f.readline()

# with open('mask.txt') as f:

    # arr02 = f.readline()

RN=list(arr01)

# arr2=list(arr02)

#print((int(".join(M), 2)))

ii=0

b=[0]*256

for i in range(0, len(M)):

    if(M[i]=='1'):

        b[ii]=RN[i]

        ii=ii+1

#print(b)

#print ii

#print(".join(b))
```

```
print(int(".join(b), 2))
#print p
gb =pow(g,int(".join(b), 2),p)
#print str(gb)
# SHA3 operatins
pt = (str(int(".join(b), 2))+FPGA_ID).decode('ascii')
H = Keccak512(pt).hexdigest()
#print pt
#print H
# Send M
conn.send(".join(M)) #int(".join(M), 2)
#M=bin(M)
#print (".join(M))
# Send gb
conn.send(str(gb)+" "+H)
#print str(gb)
# Send H
#conn.send(H)

conn.close()
print "TA"
```

The Proxy:

```
 def invmod(a, p):
    '''
    The multiplicitive inverse of a in the integers modulo p.
    Return b s.t.
    a * b == 1 mod p
    '''
    return pow(a,p-2,p)

g=3
q=   0xd0a6b524f46f5a59520d3efcba360545d911e748700ff141b7414405bcd22c0b
r=   0x1b1ba9a04575d309395ed00546339621904dafe5094ed826d081af26407f00a2
gr=  0x1302d7d599d1ec79d677e7eee28c6b565841563b17f6f3146aebc36a6382d841
a=   0x14bb28715d971d180f7055e2098e1a8a2ff67c4090afc649dc69f2424f62ccec
ga=  0x4b058bb3c58c38662bb2b8eb58534a24cba7e5194cedcb61c1f9cf5b0d890e78
gb=  0xf343a2e3522bba046a7ded8510fd2d17b6ac9faa0cb96f346a21b9668bd3164
gab= 0x6c528c1cef10ae5184e3f2a0f752b5fbb004928e80811282233b9847d3212e99

data=0x1111111111111111111111111111111111111111111111111111111111111111
rK=(ga*invmod(gr,q))%q# to be sent to the proxy
E_data=(data*gr)%q
Re_E_data=(E_data*rK)%q# to be performed  the proxy
D_data=(Re_E_data*invmod(ga,q))%q

print hex(D_data)
```

Vitae

Name :Mohammed Al-Asali

Nationality :Yemeni

Date of Birth :3/29/1986

Email :mohammedalasli@gmail.com

Address :Dhahran, Saudi Arabia

Academic Background : Mohammed received his B.Sc. degree in Computer Engineering from King Fahd University of Petroleum and Minerals (KFUPM) in 2009. In 2010, Mohammed joined KFUPM as a research assistant to pursue his M.Sc. degree in Computer Engineering. In 2013, Mohammed earned his M.Sc. in Computer Engineering from KFUPM. Then Mohammed started his career as a lecturer-B at KFUPM to pursue his PhD in Computer Science and Engineering. Mohammed Completed his PhD in May 2017.

Research Interests : Reconfigurable Computing, Cloud Computing Security and WSNs.

Publications :

M. Alasli, T. Sheltami, and E. Shakshuki, "Identifying the direction of wind in wireless sensor networks," in Procedia Computer Science, 2012, vol. 10, pp. 225–231.

M. Elrabaa, A. Hroub, M. Mudawar, A. AL-AGHBARI, **M. Al-Asli**, and A. Khayyat, “A Very Fast Trace-Driven Simulation Platform for Chip-Multiprocessors Architectural Explorations,” IEEE Trans. Parallel Distrib. Syst., pp. 1–1, 2017.

M. E. S. Elrabaa, A. A. Al-Aghbari, and **M. A. Al-Asli**, “A low-cost method for test and speed characterization of digital integrated circuit prototypes,” in 2013 Saudi International Electronics, Communications and Photonics Conference, SIECPC 2013, 2013.

S. M. Sait, F. C. Oughali, and **M. Al-Asli**, “Design partitioning and layer assignment for 3D integrated circuits using tabu search and simulated annealing,” J. Appl. Res. Technol., vol. 14, no. 1, pp. 67–76, 2016.

M. E. S. Elrabaa, A. Al-Aghbari, **M. Al-Asli**, A. El-Maleh, A. Bouhraoua, and M. Alshayeb, “A low-cost platform for the prototyping and characterization of digital circuit IPs,” Integr. VLSI J., vol. 54, pp. 1–9, 2016.