Framework de sécurité avec token pour de l'accélération cloud FPGA multi-utilisateurs sécurisée basé

Semih Ince

COLLEGE MATHSTIC
DOCTORAL BRETAGNE
BRETAGNE OCEANE

Université de Bretagne Occidentale

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ
DE BRETAGNE OCCIDENTALE

ÉCOLE DOCTORALE N° 644
Mathématiques et Sciences et Technologies
de l'Information et de la Communication en Bretagne Océane
Spécialité : *Informatique et architecture numérique*

Par
Semih INCE

TokSek : Token-based multi-tenant cloud FPGA security framework for secure acceleration

Thèse présentée et soutenue à Nokia Lannion, le 25 mars 2024
Unité de recherche : Lab-STICC UMR 6285

Rapporteurs avant soutenance :

Russell TESSIER Professeur des Universités, Université de Amherst, Massachusetts

Alain TCHANA Professeur des Universités, ENS Lyon

Composition du Jury :

Président :	Lilian BOSSUET	Professeur des Universités, Université de St-Etienne
Examinateurs :	Guy GOGNIAT	Professeur des Universités, Université de Bretagne Sud
	Maria MENDEZ REAL	Maître de conférences, Université de Nantes
	Renaud SANTORO	Docteur, Nokia Bell Labs
	Julien LALLET	Docteur, Nokia Bell Labs
Dir. de thèse :	David ESPES	Professeur des Universités, Université de Bretagne Occidentale

PREFACE

First, I would like to express my sincere gratitude to two colleagues and supervisors, Renaud Santoro and Julien Lallet, from Nokia Bell Labs. Their guidance and supervision were invaluable in the outcome of this thesis. Their support and invaluable insights helped me overcome the challenges encountered during the research process. I am truly grateful for their commitment and willingness to invest time and effort in my academic journey. I also extend my heartfelt thanks to Samuel Dubus, whose trust was important in the successful completion of this work. His encouragement and belief in my capabilities motivated me throughout my research. I am fortunate to have had such dedicated colleagues by my side at Nokia Bell Labs, providing valuable input and encouragement when needed.

My gratitude extends to David Espes from the University of Occidental Brittany and Guy Gogniat from the University of South Brittany. Their collaborative spirit and significant contributions greatly enhanced the depth and quality of this thesis. Their kindness and willingness to share their expertise played a crucial role in the overall success of this research project. I am very grateful for the positive and enriching experience of working alongside them. I want to convey my gratitude to Lillian Bossuet and Isabel Amigo for their contribution to this research work. Their feedback and inputs during our meetings were valuable, and I want to thank them for ensuring everything was on track.

I would like to acknowledge the unwavering support of all my friends in Lannion and beyond, who consistently checked in on my progress and offered words of encouragement. Their friendship provided a source of strength and motivation, reminding me that I was not alone in this academic journey. Their support made a significant impact and is sincerely appreciated.

Lastly, I want to express my deepest gratitude to my family. Their encouragement and belief in my abilities have been very impactful in my academic pursuits. Their sacrifices and constant support, especially during challenging times, have been valuable in my accomplishments. I am profoundly grateful for their enduring commitment and the support they provide.

Semih Ince, December 2023

Contents

Chapter I

Introduction

1 Context

The growing demand for computing power originates from the convergence of multiple factors. The introduction of data-intensive applications, such as machine learning, artificial intelligence, computer vision, and big data analytics, require a lot of computing resources. High performance computing is an important challenge because it allows to process and obtain meaningful insights from large datasets with great performance. Then, the evolution of technologies such as 5G, Internet of Things (i.e., IoT), and autonomous systems requires robust computing infrastructure to support the seamless integration and operation of these technologies.

Cloud computing is one paradigm for the provisioning and utilization of computing power. It offers a scalable and on-demand computing resources that aligns with the requirements of technologies with heavy workloads. Cloud computing mitigates the need for organizations to make heavy investments in dedicated hardware infrastructure, allowing them to leverage computing resources flexibly based on their specific needs. Through the deployment of virtualized environments and distributed computing architectures, cloud providers optimize resource utilization and enable users to dynamically scale their computing capabilities. This not only answers to the growing demand for computing power but also enhances efficiency, cost-effectiveness, and accessibility. Telecommunication standards are evolving and cloud computing solutions are required to support heavy computing workloads. For example, in 5G and beyond, AI/ML components are widely deployed [1]. These components require high computing power.

There are many devices like CPUs, GPUs and Field-Programmable Gate Arrays (i.e., FPGAs) for high performance computing but the latter especially take the upper hand in terms of computing power. Unlike traditional CPUs and GPUs, FPGAs offer a unique advantage by enabling users to customize and reconfigure the hardware at the gate-level resulting in a highly optimized acceleration for specific workloads.

This adaptability gives FPGAs an edge in scenarios where application-specific optimizations are important. While CPUs excel in general-purpose computing tasks and GPUs are efficient in parallel processing, FPGAs combine reconfigurability and superior parallel computing capabilities. In fact, FPGAs have significant performance advantages compared to CPUs and GPUs [2]. As a result, FPGA accelerators are becoming widely deployed by major Cloud Providers (CPs) [3], [4], [5].

In a cloud context, users require privacy, confidentiality and security. Intellectual Property (IP) and data theft must be prevented to protect cloud users' interests. In fact, data is becoming the most valuable asset of modern times [6]. It is used in various fields like AI/ML to train models or track user habits. In cloud computing, user data can be sensitive and valuable due to its origin and content. Additionally, users develop performant and costly FPGA logic (e.g., bitstream) to reconfigure the FPGA. For these reasons, protecting user data and logic inside a cloud environment is crucial for the development and the adoption of FPGA cloud computing in various technological fields.

Because a single user cannot maximize the FPGA utilization, it is more efficient to spatially divide the FPGA logic between multiple users: this is multi-tenancy. Thanks to the parallelization capabilities of the FPGA, multiple users can seamlessly benefit from FPGA acceleration. However this brings a plethora of security challenges like user isolation and access control. In fact, users must not interfere with each other and shared components like device memory must be separated and access control must be enforced.

Users need a secure remote access to the FPGA resource. Then, a direct encrypted communication channel between users and the FPGA is necessary to have secure communications. Once the secure remote access is established, the FPGA logic environment must be secured against multi-tenant and FPGA logic attacks.

2 Objective of this thesis

The objective of this thesis is to provide a multi-tenant FPGA cloud security framework called TokSek to reinforce user data confidentiality without compromising the security requirements of CPs. Low overhead and performance impact is another objective, as reinforced security measures should not prevent users from achieving efficient and high performance acceleration. To provide a secure low overhead access that supports multiple users, an adaptation of the OAuth 2 framework for FPGA cloud is proposed. It allows users to access an FPGA using an access token. The

latter is also used for access control enforcement inside the FPGA. Moreover, a hardware security module is proposed and coupled with the token-based FPGA access scheme to provide a secure acceleration environment inside the FPGA. This security module is responsible to enforce security functions like token verification and access control to shared components between tenants. To reinforce user data protection a patented solution called Linkguard is proposed. It is a zero-trust confidential channel establishment protocol to securely communicate with an FPGA located in an untrusted cloud environment. We then study a shielded enclave from the literature [7] that protects the user accelerator against the untrusted modules. We propose to upgrade the shielded module with Linkguard to address some of the identified vulnerabilities. The outline of the chapters of this work is described below.

In Chapter II, a state of the art on FPGA, cloud and security are detailed. Existing FPGA-based cloud solutions are described to provide a baseline for this research and identify drawbacks in current deployments. Because security is the principal focus of this work, mechanisms like trusted execution environments, authentication and access control are analyzed in the context of FPGA cloud computing.

Chapter III introduces a novel token-based connection method for FPGAs. A theoretical model is provided and a token-based security framework called TokSek is presented. It allows secure connection with mutual authentication mechanisms and low latency FPGA access. Additionally, a practical use case for telco providers is exposed to showcase the capabilities of this framework. TokSek is implemented on an AMD FPGA and results are presented.

Chapter IV focuses on the hardware aspect of TokSek by introducing several modules like a hardware security module inside the FPGA logic. Compared to the software approach from Chapter III, this approach aims to secure the FPGA logic by providing additional security mechanisms and offer better performance by offloading few functions to the FPGA logic. In fact, the threat model is stricter in this chapter because we consider that the processing system and the embedded operating system are compromised. These hardware security elements provide isolation and security for each tenant sharing the same FPGA. A patented security solution called Linkguard allows to establish a zero-trust confidential channel between an FPGA module and a stakeholder within an untrusted cloud environment. All these elements contribute to achieve a secure multi-tenant cloud FPGA environment that protects user data confidentiality.

Chapter V summarizes this thesis and highlights key elements that are detailed in

this work. Future works and open challenges are also described to give a perspective on future technologies that may have an impact on this work.

Chapter II

State Of The Art

1 FPGA-based cloud

1.1 FPGA-based cloud architectures

The development of demanding applications like AI/ML, video encoding and other domain specific algorithms lead to the development of accelerators. These applications are getting slower on general purpose CPUs. Algorithms and hardwares need to be more efficient to meet timing and latency constraints. Specific hardwares like GPU and FPGA are used to fulfill acceleration needs. Especially, FPGAs offer significant advantages to accelerate various applications. They have more computational power than GPUs and CPUs on most applications [2]. FPGAs are also reconfigurable to meet various acceleration needs with high flexibility. With the development of cloud services, FPGAs are already deployed by cloud providers (i.e., CP) like AWS, Microsoft and Alibaba. Depending on the need, FPGAs are deployed in the cloud under different architectures.

Cloud providers like AWS and Alibaba offer FaaS (FPGA as a Service) platforms. For example, Alibaba can provide up to four FPGAs and 88 virtual CPUs in one instance. Users get a virtual machine to access those hardwares. Cloud users can develop and reconfigure FPGAs with their custom design. Figure 2-1 shows the most common way of deploying an FPGA. The FPGA is tightly attached to a host through PCIe. The host communicates with the FPGA by using APIs and commands provided by the cloud provider. The latter has a communication and management stack inside the FPGA to execute commands sent by the host through PCIe. This type of connection is deployed by AWS F1 and Alibaba F3 instances [3], [5].

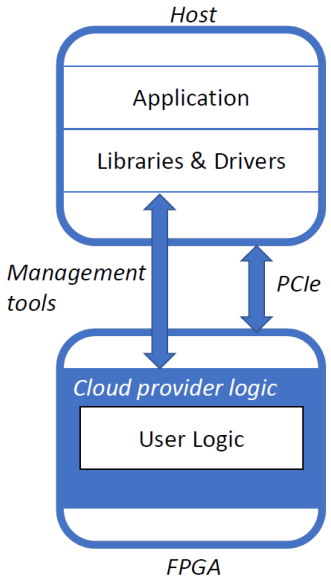


Figure 2-1 – Most common FPGA-based cloud architecture

1.2 FPGA-based cloud acceleration systems

Users can benefit from FPGA acceleration in three different ways. In FaaS model like AWS F1 and Alibaba F3, users get FPGA access to implement their own accelerator. Users have access to a virtual work environment with tools and hardware. With the Infrastructure as a Service (i.e., IaaS) model, users have a list of specific accelerators. They can choose the application they want to speed up. This is the model deployed by IBM and Microsoft Brainwawe project (neural network accelerator) [8], [9], [10]. Finally, users can benefit from FPGA accelerators with systems like Microsoft Catapult. Microsoft's web search engine Bing is accelerated with FPGAs but this is transparent to the user.

Microsoft Catapult aims to accelerate web search rankings using FPGA acceleration [9]. In this case, FPGAs are not offered to users as FaaS platforms. In fact, users can only take advantage of this system if they use Microsoft's Bing search engine. This system is totally transparent to the user. For this purpose, Microsoft opted for a Top-of-Rack (i.e., TOR) topology. They use a 2-socket server blade including two CPUs, one Network Interface Controler (i.e., NIC) and one FPGA for acceleration purposes as shown in Figure 2-2. This architecture is deployed massively in parallel to accelerate web searches. The FPGA is mainly used for acceleration in this scenario. But if an FPGA is momentarily unused, it becomes a NIC and serves as a network accelerator for the datacenter. Another example is Microsoft Brainwawe [8]. It is an FPGA-based neural network accelerator. Microsoft also uses the presented

topology for this application.

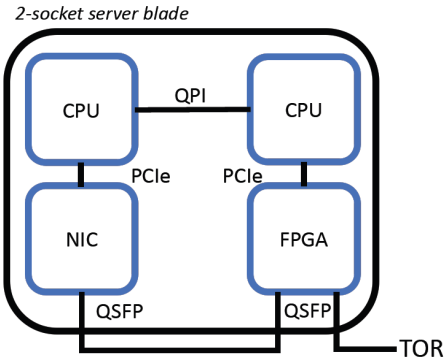


Figure 2-2 – Microsoft Catapult architecture

Finally, IBM has worked on a FPGA-based cloud architecture where FPGAs are independent from a CPU [10]. FPGAs are interconnected and attached to the data center network with a switch. The purpose of this architecture is to accelerate specific workloads like AI/ML and network encryption under a IaaS model. This means that users cannot use their custom accelerator but rather ready-to-use accelerators offered by the cloud provider.

1.3 FPGA-based Cloud without a Trusted Authority

In all solutions presented above, FPGAs are managed by the CP. The latter allocates resources and establishes security. Figure 2-3 shows the high level architecture and associated mechanisms in current cloud solutions.

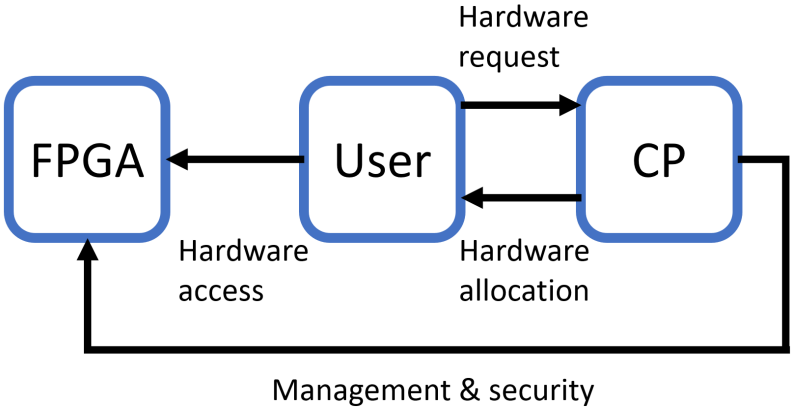


Figure 2-3 – Current FPGA-based cloud mechanisms

Upon receiving a user request, the CP creates a work environment and allocates resources. The user has access to resources through a virtualized environment. The

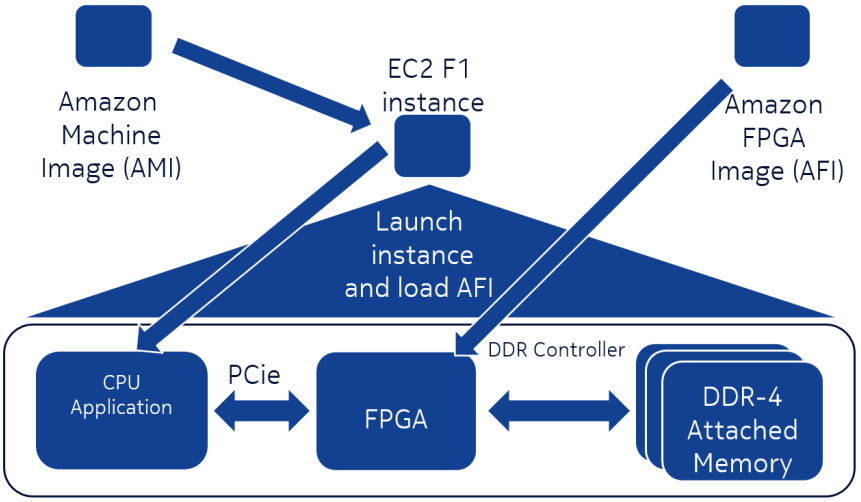


Figure 2-4 – AWS F1 instance usage.

CP can track FPGA usage and check for security issues. To achieve this, AWS and Alibaba include management functions inside the FPGA [3], [5]. Due to a lack of transparency, the CP's privileges within the FPGA are unknown. The CP can communicate with the FPGA. Thus, the CP can breach user privacy and access resources allocated to a user.

Figure 2-4 shows the mechanisms used in AWS F1 instances. The virtual machine is set up with an Amazon Machine Image provided by Amazon according to a specification selected by the user [3]. In order to program the FPGA, the user must load an Amazon FPGA Image (i.e., AFI) generated by AWS. The user must disclose their IP to program the FPGA. Before generating the AFI file, AWS checks the user IP for malicious design patterns such as power wasters and side-channel analyzers based on ring oscillators, short circuits and long wires [11] that are further described in Section 5.3. Under this scheme, the user and the CP distrust each other. The CP protects its devices against damage. Although the user desires IP protection and confidentiality, the user IP is never authenticated after the AFI file (i.e., verified user bitstream) is produced. The user has no proof that his IP in the AFI file is unmodified. Recent work has proven that bitstream manipulation is a possible way to introduce hardware Trojans into designs [12]. The user also has no proof that his IP is indeed kept secret.

In the current implementation, the user never authenticates the FPGA. He accesses a virtual machine which can access an FPGA. The lack of authentication in this scenario can be a threat as man in the middle attacks (MitM) and FPGA impersonation are possible. In MitM attacks, the attacker is placed between two communicating entities, as shown in Figure 2-5. During this attack, the virtual ma-

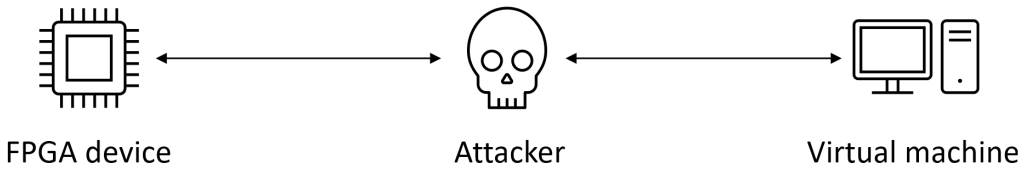


Figure 2-5 – Man-in-the-middle attack configuration.

chine sees the attacker as the FPGA and the FPGA sees the attacker as the virtual machine. In this situation the attacker intercepts the communication data, the user IP and the data.

Moreover, user IP confidentiality is lacking. To use a custom FPGA accelerator, the client must send a design file to the CP [3], [13]. To address confidentiality and isolation issues, a trusted authority (i.e., TA) can be involved as described in Section 3.2 and Section 4.3.

In this section, each presented architecture is adapted to a specific use case. Privacy measures and security assumptions are also not the same. On one hand, clients get full FPGA access under the FaaS model. End users reconfigure the FPGA with their own designs. One the other hand, users can benefit from FPGA acceleration through a web browser without requiring any technical FPGA knowledge with the IaaS model. In this work, focus is given on the FaaS model where users have access to a cloud provider FPGA. After having presented the FPGA-based cloud context, The next section focuses on Trusted Execution Environments (i.e., TEE) and FPGA enclaves. These solutions provide secure computation and create isolation against the underlying system execution environment (e.g., operating systems, concurrent programs...). TEEs and enclaves are important elements to provide secure cloud-based FPGA acceleration.

2 Trusted execution environments and enclaves for FPGA

2.1 Introduction to trusted execution environments

TEEs are hardware-based trusted execution environments located inside CPU. They provide data integrity, isolation and confidentiality for any code executed inside them. An unauthorized program running outside a TEE cannot modify the data or communicate with a program running inside a TEE. Most common TEEs are Intel SGX [14], Arm TrustZone [15] and RISC-V Keystone [16]. These TEEs are available

for their respective processors. For example, an Arm CPU cannot take benefit from SGX or Keystone. Each TEE solution aims for similar achievements like isolation and integrity but they do it differently.

There are mainly two types of FPGA architecture and one of them is FPGA SoC (i.e., System on Chip). They include a Processing System (i.e., PS) with an embedded CPU and can have various other components like on board memory, Graphics Processing Unit (e.g., GPU) and various interfaces (e.g., UART, USB, CAN...). On the PS side, one can set up an embedded OS and use the board as an embedded device by using vendor tools. The PS can execute software programs by leveraging the CPU and the tools available in the embedded OS. Most importantly, the PS can communicate with the other part of the SoC : the Programmable Logic (i.e., PL). The latter is a matrix of programmable hardware resources (e.g., look-up tables, flip-flops...) which can implement hardware functions.

The second FPGA architecture are accelerator cards that do not have a PS. Those devices are solely constituted of programmable hardware (i.e., PL) in order to reach low latency and high throughput acceleration. Accelerator cards are connected to a host machine with a PCIe interface. The host machine sends commands and jobs through the PCIe interface to accelerate a specific function that is deployed inside the accelerator card. Those devices are often used in large scale data centers to reach higher performance on critical workloads. The AMD Alveo series or the Intel Stratix FPGAs are some examples of accelerator cards [17], [18].

For their FPGA SoC, Intel and AMD use Arm CPUs like the Quad Arm Cortex-A53 for their PS [19], [17]. Consequently, Intel and AMD have TrustZone technology enabled on their PS. Accelerator cards without a PS cannot leverage TrustZone or SGX technology because they do not have a CPU. To benefit from a TEE, accelerators can implement a RISC-V CPU inside the PL to benefit from the Keystone TEE [16], [20]. However, this aims to protect the software execution inside the RISC-V processor and the hardware logic still remains unprotected. RISC-V is an open standard for an ISA (i.e., Instruction Set Architecture) and many RISC-V implementations exist due to its customization capabilities [21], [22], [23].

2.2 Major frameworks for trusted execution environment

Intel Software Guard Extensions (i.e., SGX) is a hardware-based TEE that aims to provide secure code execution and data protection. At its architectural core, SGX introduces secure enclaves and isolated memory regions within the CPU [14], [24],

[25]. Enclaves enable the execution of critical code and the handling of sensitive data with a high degree of security and confidentiality. One notable accomplishment of SGX is its ability to enhance software execution security and protect data integrity and confidentiality, even if the underlying operating system or hypervisor may not be entirely trusted [14]. The notion of hypervisor is further explained in Section 5.2. Some features of SGX are memory isolation mechanisms, encryption techniques for enclave memory pages, and a robust remote attestation system for enclaves. SGX ensures that code executing within its enclaves remains confidential, immune to tampering, and secure against various threats. As a result, SGX plays a vital role in securing applications and workloads in areas such as secure cloud computing, confidential data processing, and secure code execution. To provide these security achievements, SGX relies on software attestation mechanism. The latter provides cryptographic evidence of an enclave's identity and integrity, enabling remote parties to verify the enclave's security status. SGX also supports secure data sealing and unsealing, allowing sensitive information to remain protected, even when stored outside the enclave. SGX is particularly interesting for FPGA cloud computing because it uses the concept of a Trusted Authority (i.e., TA) to provide an external remote attestation service. Later in this chapter, the benefits and shortcomings of a TA-based solution is discussed.

Arm TrustZone is another notable hardware-based TEE. Its architecture is based on the concept of secure and non-secure worlds, enabling the execution of sensitive code and data within a fortified, isolated environment [15], [26]. The same way as SGX, TrustZone is also providing software security and data confidentiality even if the underlying operating system or hypervisor is not fully trusted. This is accomplished by robust memory isolation and peripheral protection mechanisms [27], secure boot processes, and cryptographic hardware acceleration. TrustZone ensures that code running within its secure world remains confidential, impervious to tampering, and resilient to various threats [28]. Consequently, TrustZone plays an important role in providing the security of applications and workloads in areas such as mobile devices, embedded systems, and Internet of Things (i.e., IoT) devices. Hardware-enforced memory separation ensures that code and data in the secure world remain inaccessible to the non-secure world. TrustZone's secure boot process guarantees the integrity of the system's boot sequence and firmware, preventing unauthorized modifications. Furthermore, cryptographic hardware acceleration enhances the performance of cryptographic operations and secures sensitive data. TrustZone also supports secure key storage and cryptographic operations within the secure world, further strengthening its security. These mechanisms mitigate threats

such as unauthorized access, code injection, and data breaches while creating isolation against the non-secure world.

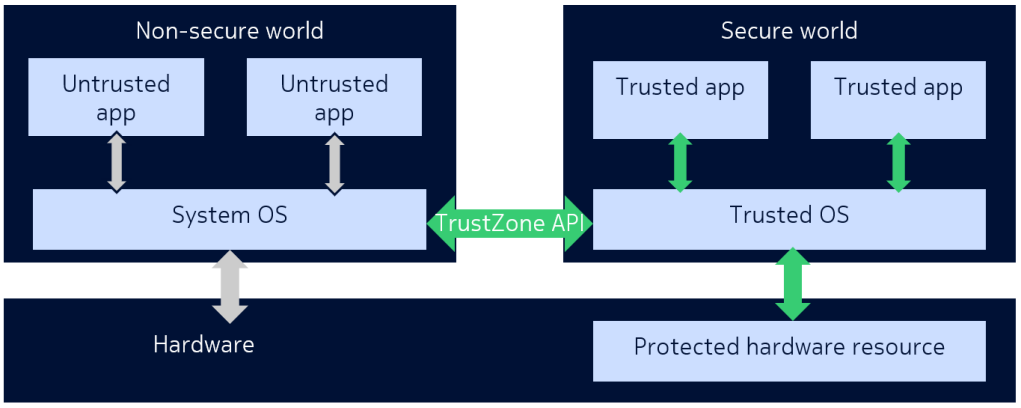


Figure 2-6 – Arm TrustZone architecture

Keystone is an open-source TEE framework compatible with RISC-V architecture [16], [20]. It provides a secure enclave for code execution and data protection within its architectural design. Keystone enclaves have isolated memory regions to ensure the confidentiality and integrity of sensitive code and data. One notable aspect of Keystone is the customization, allowing developers to use security features of the framework for their specific use cases. The framework's compatibility with the RISC-V architecture enhances its accessibility, making it an option for projects seeking a balance between security and adaptability. As a key component, Keystone uses its Physical Memory Protection (i.e., PMP) primitive to specify protections on memory regions of the RISC-V operating system. An unauthorized program or master (e.g., AXI master-slave communication protocol) is unable to access specific memory regions [20]. Another notable feature is the execution of a trusted Security Monitor (i.e., SM) using the machine mode, establishing security boundaries without engaging in resource management. There are three levels of privilege in Keystone : user mode, supervisor mode, machine mode. Machine mode has the highest privilege and only the SM has access to it. Enclaves in Keystone operate within dedicated physical memory regions, each equipped with its own supervisor-mode runtime (i.e., RT) component for virtual memory management and enclave-specific mechanisms. This enclave architecture enables secure and independent implementation of enclave-specific features by their runtime components, while the security monitor oversees hardware-enforced guarantees. The enclave runtime focuses solely on essential functionalities, communicating with the security monitor, managing host interactions through shared memory, and servicing the Enclave user-mode Application (i.e., EApp).

MultiZone FPGA is also an open-source and standard-based TEE specifically designed for RISC-V processors [29]. This TEE offers hardware-enforced separation for multiple secure domains, granting full control over data, programs, and peripherals without requiring additional IP blocks or firmware modifications. It is a versatile TEE that can create a policy-driven security environment for RISC-V, applicable to a wide range of devices from single-core IoT to multi-core Linux applications. Key components include the lightweight MultiZone nanoKernel, the secure inter-zone communication infrastructure MultiZone Messenger, MultiZone Configurator for secure boot firmware generation, MultiZone Secure Boot. MultiZone nanoKernel is a lightweight kernel designed for enforcing policy-driven hardware separation of RAM, ROM, I/O, and interrupts. This kernel establishes secure and isolated environments between different execution zones. The MultiZone Messenger is another critical component, serving as a communications infrastructure that facilitates secure message exchange across zones without shared memory. The MultiZone Configurator is useful in combining linked zone executables with policies and the nanoKernel to generate the secure boot firmware image, allowing developers efficient configuration of zones based on specific security requirements. Additionally, the MultiZone Secure Boot module enhances security with a two-stage secure boot loader, employing cryptographic algorithms such as SHA-256, SHA-512, and ECC to verify the integrity and authenticity of the firmware image. These modules provide an adaptable framework, enabling the construction of secure and isolated Trusted Execution Environments (zones) for RISC-V applications.

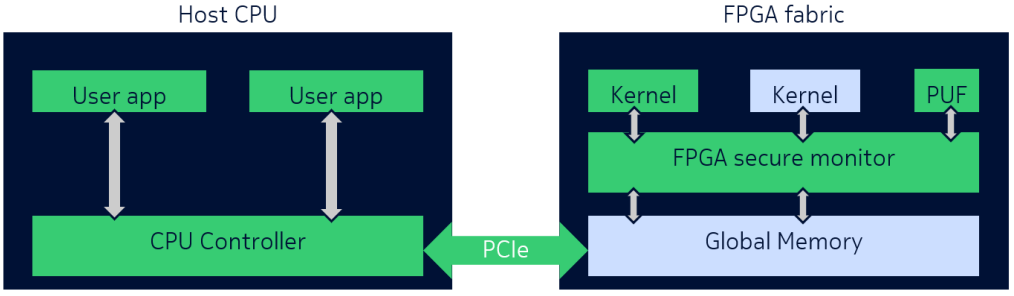


Figure 2-7 – Architecture of SGX-FPGA : an extension of Intel SGX to FPGA platforms. Green modules are trusted

Some solutions in the literature leverage the reconfigurable aspect of the FPGA to provide a secure and isolated enclave solution for specific applications. Because CPU-based TEEs do not provide FPGA support, researchers looked forward to extending existing TEE frameworks to FPGAs.

SGX-FPGA is a solution that aims to extend SGX capabilities to FPGA en-

vironments [30]. This is a CPU-FPGA heterogeneous solution to bridge software and hardware enclaves to protect the data and the computation in both worlds. They combine the use of a SGX-enabled host CPU with an FPGA that is connected through a PCIe interface. A CPU controller located inside the SGX enclave and an FPGA secure monitor in the programmable logic are proposed. They are both trusted. All EApps are connected to the CPU controller. The latter is responsible of the PCIe interface and FPGA communication. The CPU controller is responsible of cryptographic operations for the communications with the FPGA secure monitor. On the PL side, the FPGA SM also does cryptographic operations and it has its own on-board attestation. To establish isolation between the FPGA and the user enclave, the EApp and the CPU enclave authenticate themselves with the remote attestation server. Both of them receive a certificate for a successful authentication with the remote attestation server. Then, the EApp and the CPU enclave can authenticate each other. To setup secure communication, they both generate and exchange encryption keys. The CPU enclave also authenticates and exchanges keys with the FPGA SM. The latter generates keys with a PUF (i.e., Physical Unclonable Function) which is described in Section 4.2. There are in total two generated keys and two authentications to establish secure communication between an EApp and the FPGA logic. However, a drawback of this solution is the high logical resource consumption. This design needs 175k LUT which is approximately 25% of an AMD Virtex 7 FPGA. In a multi-tenant cloud FPGA context where multiple users share the same accelerator device, this solution may either create congestion in the FPGA logic or lead to a greatly reduced number of possible tenants. It is notable that a highly congested FPGA fabric creates security vulnerabilities. Some of them are further discussed in Section 5.3.

2.3 Security vulnerabilities of trusted execution environments

Despite aiming to provide secure execution and isolation, TEE security vulnerabilities exist.

For example, previous works [31], [32], [33] describe vulnerabilities found on Intel SGX TEE. As described earlier, SGX relies on untrusted software to run trusted enclaves. SGX is prone to passive memory mapping attacks because the enclave page table and address Translation Look-aside Buffers (i.e., TLB) are managed by the untrusted OS [33]. Page tables store the mapping between virtual and physical address. With passive memory mapping attacks, the untrusted OS can identify the

memory region used by an SGX enclave. However, active attacks are not possible because page table content is encrypted and its integrity is protected.

SGX is also vulnerable to Spectre attacks. It is a micro-architecture side-channel vulnerability based on speculative execution. The latter is an optimization technique where the CPU executes tasks before they are needed. [32] shows that the branch prediction of the SGX enclave can be influenced by programs outside the enclave. By doing that, internal register and cache content can leak outside the SGX enclave, suppressing the confidentiality promises of Intel SGX. Encryption keys and attestations stored inside and SGX enclave are successfully stolen [32]. There are also a plethora of cache attacks, like timing attacks which allow to steal enclave protected data [34], [35], [36].

Arm TrustZone also has some vulnerabilities reported in the literature [33], [37], [38]. Compared to Intel SGX, TrustZone does not have memory mapping vulnerabilities because enclave page tables are managed by the secure world and a trusted software. Normal world and secure world separation is providing more isolation compared to SGX which runs on untrusted software. However, TrustZone has the same cache vulnerabilities of SGX [33] [37]. Additionally, Arm CPUs are used in FPGAs and this leads to a greater attack surface. For example, AMD UltraScale+ architecture is a SoC architecture using Arm CPUs. Work [37] shows that the Accelerator Coherency port (i.e., ACP) is a vulnerability for TrustZone isolation. The ACP port is a cache coherent AXI port used for applications where PS-PL communications are frequent. Moreover, using a hardware Trojan, authors demonstrated a Direct Memory Access (i.e., DMA) attack, compromising isolation mechanisms and TrustZone secure boot procedure [37], [39]. The attacker can reprogram the eFuse and modify the encryption key and key hash that are required to configure the FPGA logic. This can be achieved by using the ACP port that can bypass the Xilinx Memory Protection Unit (i.e., XMPU). After the modification of the eFuse, the attacker can load unauthorized logic into the FPGA bypassing TrustZone secure boot. Previous work [38] also exposes vulnerabilities of Arm TrustZone in FPGA SoC. This work takes advantage of the AXI communication to disrupt device isolation. By adding intermediate modules like a FIFO in the AXI interconnect block, or modifying response signals, an attacker can leak protected data, execute privilege escalation or denial-of-service attacks.

Compared to SGX and TrustZone, RISC-V implementations using Keystone are considered safer due to continuous development from the community. RISC-V implementations are studied and vulnerabilities are reported from the community. Some

RISC-V implementations get out-of-order (outdated) due to security vulnerabilities or insufficient feature/performance [40]. To improve security, work [40] proposes TEESec. It is a pre-silicon software for vulnerability discovery for TEEs. Security vulnerabilities of out-of-order RISC-V implementations like BOOM [41] and XiangShan [42] are exposed. They found data and metadata leakage case in both implementations. In both implementation, enclave data leaks and Keystone secure monitor data leaks have been found. These leaks are originating in missing PMP permission verification for cache mechanisms (e.g., cache pre-fetcher) and Meltdown-type vulnerabilities [40], [43]. However, because each RISC-V is an open-standard for CPU design, each implementation is different. Thus, security vulnerabilities and features can be different for each implementation. RISC-V XiangShan [42] and Boom [41] do share similar vulnerabilities but they also have their own ones. For example, Boom [41] has residual data leakage after enclave destroy whereas XiangShan [42] is secure in that regard.

Few mitigations exist for various TEE vulnerabilities described above. For RISC-V and Keystone, flushing caches seems to be possible mitigations for both Boom [41] and XiangShan [42] but this may affect performance [40]. For FPGA SoC using an Arm CPU, mitigations exist to secure the vulnerabilities created by the AXI ACP port. If one intends to use the ACP port to benefit from L2 cache, a specific isolation module can be developed for the ACP [37]. This module can enforce specific rule sets according to its configuration before communicating with the ACP port. Additionally, work [38] proposes two software memory controllers to prevent the use of AXI interconnect and memory interconnect modules while enforcing memory rule sets. However for SGX, the architecture of the CPU cannot be modified to fix security issues. Some security issues located in the CPU architecture can be fixed by microcode patch. Microcode modification allows to change the CPU behavior and modify the task execution. For example, SGXpectre [32] can be fixed with such a patch. Additionally, vulnerabilities like CacheQuote [36] can be fixed by changing the enclave service framework to support higher level of enclave provisioning mechanisms [31]. And finally, a great number of reported attacks can be fixed using adequate compiler optimization techniques and application design methodology [31].

The current section described existing TEE solutions and how they can contribute to the security of FPGA-based cloud computing. In this section, TEE frameworks are analyzed, security vulnerabilities are exposed and possible mitigations are detailed. The following section focuses on authentication in FPGA-based cloud computing. It is one of the most important security features for FPGA-based cloud computing. It allows to identify acting entities (i.e., users, cloud providers), targeted devices and

custom user designs. In a cloud context, authenticating users and devices allows the cloud provider to set up authorization and access control. For the user, authentication is a proof that the work environment (e.g., the allocated resources) is genuine and identified.

3 Authentication techniques

Authentication is a mean to verify the identity and the authenticity of objects or subjects. Let's consider Alice and Bob communicating over an unsecured line. In order to mutually authenticate themselves they can have a shared secret like a passphrase (i.e., password) and verify that each other know it. They could also share their own identifier provided by an entity they both trust. Alice would verify Bob's identity provided by the trusted entity. If the trusted entity confirms that the identity provided is valid, Alice can trust the person pretending to be Bob. They would continue to communicate securely using public key cryptography.

In this section, various authentication techniques are detailed. Some techniques allow users to directly authenticate each other. Other authentication mechanisms leverage a trusted third party to authenticate users.

3.1 Direct authentication

Authentication mechanisms often involve digital signatures and hash functions. The latter is a one way function which maps $h : X \longrightarrow Y$ where $|X| = n$, $|Y| = m$ and $n > m$. It is relatively easy to compute the result with a given input. But computing the input with a given output is extremely difficult. SHA (i.e., Secure Hash Algorithm) is a popular hash function. In fact, a hash algorithm guarantees integrity of data but it does not authenticate the sender/receiver [44]. There are no shared secrets in hash functions to achieve authentication. This means that an attacker can still construct a message with a correct hash and send it. The receiver will see this message as valid. An attacker only needs to know which hash function is used. As $|X| > |Y|$ it is possible to brute force the hash algorithm. Collision attacks [45] allow to find the same hash output with a different input using a brute force algorithm. There are different ways to authenticate an entity. Technical details and features of different authentication techniques are detailed below.

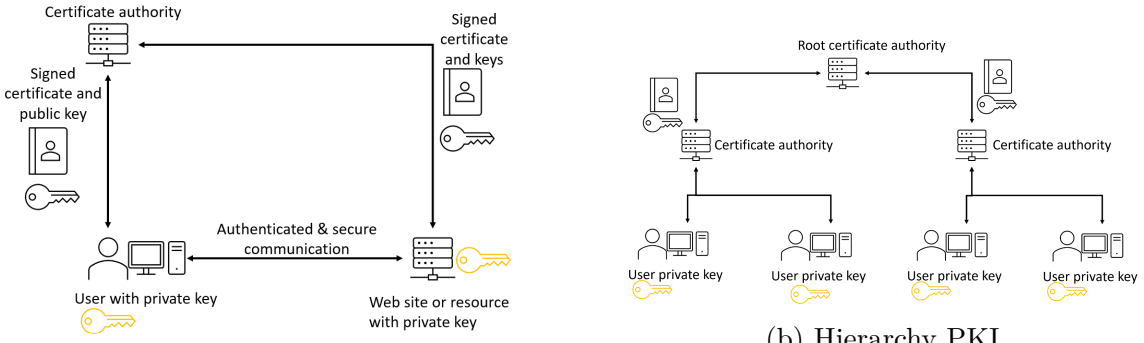
In order to authenticate a sender, Message Authenticated Code (i.e., MAC) algorithm is one possibility. It uses a shared secret to create message digests. You can

only create authentic and valid messages if you have the shared secret. In order to authenticate and verify message integrity, HMAC (keyed-Hash Message Authenticated code) functions are optimal. HMAC is the use of MAC (Message Authenticated Code) with a hash function. HMAC and MAC algorithms serve the same purpose. HMAC is a specific implementation of MAC which includes a cryptographic hash functions. Thus, HMAC has stronger security properties than a standard MAC.

Authenticated Encryption (i.e., AE) is a way to authenticate the data and the sender. AE is a block cipher mode which provides encryption as well as integrity and authentication of the data. Common AE algorithms are AES-CCM and AES-GCM [46]. AE can be seen as an encrypted HMAC because it involves a shared secret, a hash function and finally an encryption method. The most common schemes of AE is Encrypt-then-MAC (EtM), MAC-then-Encrypt (MtE) and Encrypt-and-MAC (E&M) [47]. Each method is slightly different from one another but they achieve the same objective. EtM method is considered to have the highest level of security between the three choices [48]. E&M and MtE are also secure but they require few modifications to be strongly unforgeable [48]. AE is also tightly linked to TLS (i.e., Transport Socket Layer) and in every application where confidentiality and integrity is required. AE algorithms are mandatory in TLS, encryption and hash functions are used in every communication. The choice of the algorithms depend on the negotiation between the server and the client [49].

3.2 Authentication involving a third party

Certificates are another way to authenticate a user. Information like public keys, name and organization are present in the certificate. Often, certificates work under a Public Key Infrastructure (i.e., PKI) scheme. Under this scheme, a Certificate Authority (i.e., CA) signs user certificates and generates public keys as shown in Figure 2-8a. CAs are trusted anchors, they sign certificates to indicate that the user's identity is verified. Generally, there are other entities like the Registration Authority (i.e., RA) in PKI. There are various architectures and entities for PKI systems [50], [51]. Figure 2-8b shows a single rooted hierarchical PKI structure. A root CA (i.e., trust anchor) certifies multiple CA. A certificate signed by a CA can be verified if it can be traced back to a trust anchor (i.e., root CA). This scheme is widely used by web browsers. Holding and updating a list of root CA is enough to verify a certificate. Nowadays, PKI is everywhere. The most notable system using PKI is TLS. It allows to achieve authentication and secure communication over HTTP (i.e., HyperText Transfer Protocol) [49]. Through a client-server communication,



(a) Basic PKI

(b) Hierarchy PKI

Figure 2-8 – Public Key Infrastructure mechanisms

entities exchange their respective certificates for verification. This is mutual TLS authentication and it is an optional mechanism. The certificate is signed with the CA's public key. To verify the certificate, a user needs to contact the CA and send it for verification. If the certificate is valid (i.e., not revoked) the CA gives a positive response to the user. Each entity is identified and associated to a certificate. X.509 certificates are popular in TLS. They include information like issuer name, subject name, public key information, validity period etc. For each future communication, the certificates can be verified to achieve authentication. Upon successful verification under TLS 1.3, both entities proceed to communicate by creating a shared secret with DHE (i.e., Diffie-Hellman Ephemeral) or ECDHE (i.e., Elliptic Curve Diffie-Hellman Ephemeral). In TLS 1.2, the most common method remains RSA. In recent works on cloud FPGA, the expression "Trusted Authority" (i.e., TA) is more common. The role of the TA is similar to a CA. Works like [52] and [53] take advantage of a TA to generate FPGA certificates for authentication and encryption. Isolation between the user and the Cloud Provider (i.e., CP) is reinforced to offer better confidentiality. Security critical functions like authentication and certification are done by the TA instead of the CP. If a third party TA is not present in the FPGA allocation scheme, the CP can control every security mechanism and every step of the FPGA sharing. Hence, the CP can have access to user IP, data and allocated resource. To the best of our knowledge, no commercial cloud provider takes advantage of a TA for cloud FPGA services at the time of writing.

Other specific tools like SAML (Security Assertion Markup Language) and OAuth2 also exist. These are authentication and authorization tools commonly used with HTTP environments. SAML is a framework based on XML. It allows to share identity and security information to access other domains [54]. On the one hand, SAML allows to access multiple corporate tools and domains under one identity. On the

other hand, OAuth2 allows to make authenticated HTTP resource request [55]. As shown in Figure 2-9, OAuth2 is a protocol involving (most of the time) four entities : a user, a Resource Owner (RO), an Authorization Server (AS) and a resource server. The user negotiates an agreement with the RO to access its resources. The RO specifies the scope (i.e., rules and limitations) of the resource sharing. If the user gets the authorization from the RO, the user can ask the AS to generate an access token. The user can use the access token with the resource server to get access to the resources shared with him. This scheme is lightweight and practical when a user needs to access tools quickly without successive log in. OAuth2 is much more focused on authorization, whereas SAML is more about authentication. Authentication do exist in OAuth2, but the solution to authenticate a user is out of the standard's scope and left to the developer. Later in this work, an adaptation of OAuth2 for confidential cloud FPGA sharing is detailed [56].

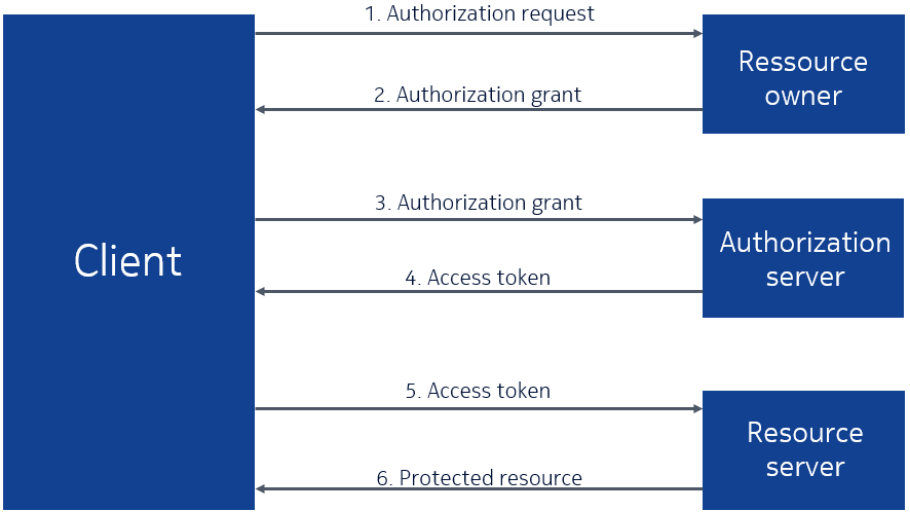


Figure 2-9 – High level view of the OAUTH2 protocol

4 Authentication in FPGA cloud environments

Current cloud providers like Amazon, Alibaba and Huawei offer FPGA access through virtual machines. Figure 2-10 shows the most common way to access an FPGA deployed on the cloud. Users are authenticated two times to access an FPGA. On step 1, the customer authenticates himself using his AWS account to request a VM with FPGA access. On step 2, the CP provides a VM access to meet the user's hardware requirements. On step 3 the customer logs into the VM and gets his FPGA access on step 4.

4.1 Bitstream authentication for FPGA cloud

A bitstream is a configuration file produced by an FPGA development software like AMD Vivado or Intel Quartus. To run an algorithm, a bitstream is loaded inside the FPGA. The bitstream targets a defined configurable area and can run until device reset or reconfiguration. Bitstream authentication is an important aspect of IP protection in FPGAs. Users want to configure the device with their custom design but keep their IPs private. By achieving bitstream authentication, the FPGA device is able to verify if the bitstream comes from an authenticated source. Bitstream integrity is also checked and any modification on the bitstream can be spotted. Therefore, it is not possible for an attacker to configure the FPGA with a malicious IP. It is also not possible to implement hardware Trojans in valid custom user designs at a bitstream level.

In [46], authors propose a solution to achieve bitstream authentication, integrity and encryption by implementing a hardware block cypher AE scheme. From an FPGA resource point of view, block cypher algorithms like AES-CCM or AES-GCM are much more efficient than standard AES+HMAC algorithm. Authors implemented a compact AES-CCM algorithm with four times less logical resource at the cost of three times less throughput compared to AES+HMAC algorithm. Authors of [57] implemented AES-GCM and also compared it against AES+SHA. Their design is 25% faster than AES+SHA and with slightly less logical resource used.

In [58], authors proposed a secure protocol for secure remote bitstream update. They particularly aimed to prevent spoofing and replay attacks. Spoofing is the replacement of a genuine data transfer with other data. Replay attacks are done by catching a data transfer and replaying it anytime over the network. In their scheme, bitstream version is controlled and only the system designer can update the active bitstream. For authentication, the system designer, the bitstream and the FPGA are authenticated through block cipher (i.e., AE).

Cloud providers like AWS and FPGA vendors like AMD propose FPGA bitstream marketplaces. A user having access to an FPGA cloud service can buy a third party IP and implement it in the resources. In [59], authors propose a solution for this scenario. The user needs to share an FPGA identifier when buying an IP from the software vendor. The software vendor sends the requested IP and the FPGA identifier to the FPGA vendor. The latter is able to identify the target device using the FPGA identifier. The FPGA vendor then encrypts the IP with the shared secret associated to the FPGA identifier. The FPGA vendor has a database of shared secret associated to every FPGA that they sold. Bitstream integrity and authenticity is

achieved with a keyed-hash message authentication code (HMAC). The encrypted IP is sent back to the software vendor who verifies integrity and forwards the IP to the user. The bitstream is bounded to a single device because of device targeted bitstream encryption.

Bitstream authentication mechanisms often involve the FPGA and a shared secret because a bitstream targets a specific device. To guarantee IP integrity and overall security, FPGA authentication is important to ensure the authenticity of the hardware endpoint receiving the IP because sensitive user data will be processed inside the FPGA.

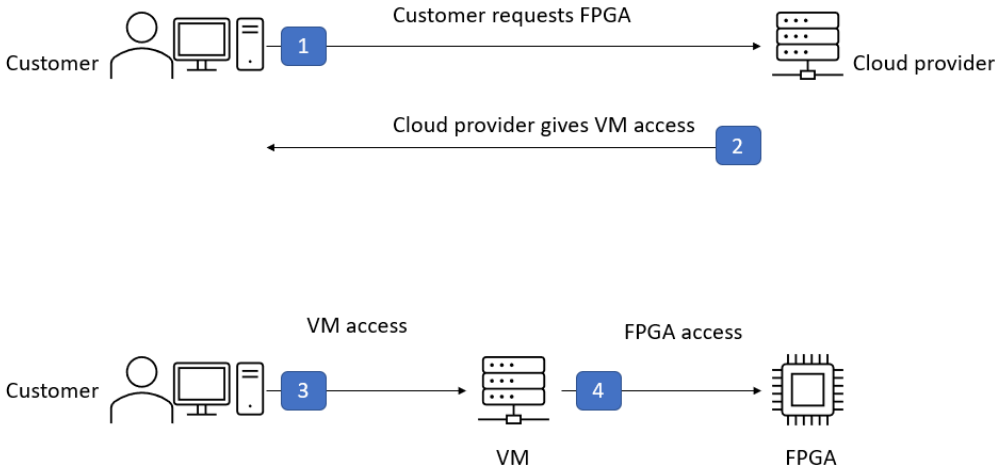


Figure 2-10 – Common FPGA access scheme

4.2 FPGA and user authentication

Authenticating the FPGA device is critical in remote FPGA computing. As users do not have physical access to the device, they need to ensure that they are communicating with the appropriate FPGA. As stated above, AE is one way to authenticate a device. With a shared secret and a block cipher it is possible to mutually authenticate the user and the FPGA to achieve message integrity altogether. Previous works implemented a compact 32-bit AES-CCM (i.e., CBC-MAC and counter mode encryption) [46], and a 3DES block cipher for FPGA authentication [58]. Shared secrets or encryption keys can be implemented in secure memories and one-time programmable memories. For SoC (System on Chip) platforms, Trusted Execution Environment (TEE) like Arm TrustZone and Intel SGX can be used to offer secure memory and security sensitive computation.

It is also possible to share a secret without storing anything inside a secure

memory. Physical Unclonable Functions (PUF) [60] take advantage of the physical randomness of the FPGA silicon that is unintentionally created during the manufacturing process of the semiconductor. This randomness mostly affects rising and falling times of signals. As the semiconductor physical characteristics cannot be reproduced, each FPGA device behaves differently to PUF designs. A specific input to the PUF (i.e., a challenge) produces an associated output (i.e., a response). The output for a given input is unique to the device. The same PUF implemented on multiple devices will give different results. If a PUF design is characterized and challenge-response pairs are saved, it is possible to identify deployed FPGAs remotely. One would have to send a challenge and read the response to validate the FPGA authentication.

But PUF designs are also not ideal. They are especially vulnerable against machine learning and modeling attacks [61]. When enough challenge-response pairs are collected, it is possible to predict the response for a given challenge. PUFs are also sensitive to noisy environments. The temperature and the device voltage have a significant impact on the PUF responses [62]. If they are not stable, the voltage can fluctuate enough to create a Denial of Service (DoS) attack on the PUF. As a result, authentication requests are not possible because the response for a given challenge is not going to be reproduced as expected. As a requirement, a secure PUF-based authentication protocol should be resilient against machine learning attacks and noisy environments. Additionally, it should be resilient to brute force. The PUF response must be long enough to make this type of attack more difficult. Depending on the design, PUF responses only produce one or two stable fully random bits for one challenge. The method used to expand the PUF response or the authentication protocol itself must not create new possible attacks. For example an attacker should not be able to send challenges to a PUF design. There should be a mutual authentication between the PUF design and the PUF user, otherwise the PUF can be used by anybody and all the challenge-response pairs can be stolen. The mutual authentication should not be solely based on a shared secret. Not storing a shared secret is the biggest advantage of a PUF. If mutual authentication is based on a shared secret, using a PUF becomes irrelevant. To protect the PUF design, the FPGA should authenticate the user. Upon successful user authentication, the FPGA should allow the PUF usage. As for now, PUFs need further research to be an actual widespread security solution.

Other FPGA authentication techniques are also possible. Other works use certificates and PKI [52], [53]. The idea is to use public key cryptography and certificates to communicate outside the FPGA and take advantage of the trusted authority present

in the scheme. The following section gives further details on those mechanisms.

4.3 TA-based solutions for FPGA and user authentication

A Trusted Authority (i.e., TA) is often used to create isolation between the user and the cloud provider to reinforce confidentiality and privacy. In [52], authors worked towards creating an FPGA enclave for cloud computing with the implication of a TA. The aim is to reinforce user IP security, authentication and privacy in public FPGA clouds while supporting multi-tenancy. An on board key hierarchy is also used with a Device Unique Key (i.e., DUK) as a root. The DUK is controlled by the TA and it is deterministically generated using a PUF. In order to establish security, the DUK is derived multiple times for different purposes (e.g., bitstream encryption, enclave specific keys). The TA can decide to provision a new DUK and update all the other keys derived from it. The solution for FPGA authentication proposed here is an SGX-inspired attestation mechanism endorsed by the TA. The latter offers services like bitstream certification and boot code authentication. In order to provide a secure FPGA environment, security critical components like the device unique key and bitstream loader are controlled by the TA. This way, the TA controls the root of the key hierarchy and can update/revoke keys depending on security threats. For enclave communication, authors decided to implement public and private keys as well as a certificate. Keys are used to create a secure channel between an enclave and a user, and the certificate is used to authenticate the enclave and the running design.

In [53], the authors aimed to protect user data and bitstream from the CP using a TA. A TA must install and manage encryption keys inside FPGAs. The FPGA is added into a PKI, it has a public and private key to establish secure communication with cloud users. To protect his bitstreams, the client uses the FPGA public key to negotiate a symmetric encryption key known as the session key. The latter will be used with an AES core to provide data confidentiality. The integrity of the data is ensured with a SHA core. Because encryption keys are managed by the TA, the user has a secure method to communicate with the FPGA while being isolated from the CP. This solution has few security limitations. The user is never authenticated, and the FPGA is indirectly authenticated with the encryption keys installed by the TA. This means that if an attacker successfully retrieves the keys stored inside the FPGA, the whole content of the device is compromised. This creates other attack vectors on key renewal and distribution mechanisms. Finally, multi-tenancy is not supported with this method.

A solution to protect user data and bitstream from the CP is presented in [53]. The FPGA user is not authenticated and user-FPGA authentication never happens. The FPGA is indirectly authenticated thanks to encryption keys deployed by the TA.

Work [63] is another example of FPGA cloud architecture using a TA. The main goal of this work is to provide an architecture with a faster setup time than a full virtualization scheme (e.g., virtual machines). This solution is a secure FPGA framework to protect the privacy and integrity of user data and IP in FPGA clouds. Multiple security components are deployed inside the FPGA to provide a secure environment. The TA implements accelerators for AES and SHA and other specific IPs like Physical Unclonable Functions (i.e., PUFs). To get access to a resource, the user gets an FPGA serial number by the CP. Then, the user sends it to the TA to retrieve FPGA authentication credentials and a session mask which is known by the FPGA. The user forwards the mask and his own portion of the mask that he generated himself. The FPGA uses the mask to calculate authentication credentials by using a modular exponentiation algorithm. Then, the results are sent to the user for verification. If the received credentials are identical to the credentials that the TA shared with the user, the FPGA is authenticated. The user authenticates the FPGA by comparing the PUF output hash with the hash received by the TA. If authentication is successful, a shared secret is established between the user and the FPGA and a secure channel is created between the user and the FPGA. With the secure channel and the session key, the user can securely communicate with the FPGA and send encrypted bitstreams. However, this solution has a security limitation. User authentication is not achieved in this scheme. In fact, only FPGA authentication is mentioned in this work. The FPGA never authenticates the user during the described scheme. Therefore, a malicious user can impersonate a user and access the FPGA by stealing the FPGA authentication credentials that the legitimate user received from the TA. The user gets FPGA access after the session key is set-up. As a consequence, the session key is crucial for the FPGA access security and must be kept secret. A multi-tenant mechanism is not described in this work. The secure channel is established with an FPGA identifier and a session mask. For security reasons, tenants cannot use the same inputs to generate a secure channel with the FPGA.

In [64], authors are able to leverage CPUs and TEE to achieve security for an FPGA-based cloud. They provide IP confidentiality for the user while allowing the CP to secure its FPGA devices and cloud infrastructure with the implication of a TA (i.e., FPGA Vendor). In fact, while the CP verifies the user IP inside a TEE,

the TA is responsible of the FPGA shell and various attestations (e.g., FPGA shell, application). Indeed, the authors give a great emphasis on authentication. The user needs to prove their identity to the FPGA in order to prevent user impersonations. Then the user can authenticate the active FPGA shell. This way, the user knows that the security elements of the device are not compromised. Lastly, the user's bitstream is authenticated by the FPGA and its integrity is verified. This level of security is not only optimal but also mandatory when it comes to FPGA cloud computing. Both the user and the cloud provider have expensive technology at stake. Taking advantage of a TEE inside the processing system of the FPGA for design verification has great advantages for confidentiality and security, but this can create great latency. Especially in a multi-tenant context, the user design verification from the TEE can be a serious bottleneck.

In this section, the emphasis is given on authentication and FPGAs in a cloud context. There are three major elements to authenticate in that regard: the FPGA, the bitstream, and the user. Various techniques like AE, certificates and PUF designs are described. The advantages of a TA-based authentication scheme is also detailed. It allows to create isolation between the user and the CP. In the next section, FPGA hardware deployment architecture, virtualization and multi-tenancy solutions are detailed.

5 FPGA architecture and multi-tenancy for cloud computing

5.1 Lack of efficiency in current FPGA cloud deployment

Currently in FPGA cloud services, one FPGA is allocated to one customer. Often, high-end devices are deployed for acceleration services. For example AWS offers AMD Virtex UltraScale+ VU9P FPGA. This device has 1.182 million LUTs and approximately 2.6 million flip-flops (FF). As shown in Table 2-1, authors of [65], [66], [67] developed a neural network accelerator. For example, [65] used 131,042 LUTs and 113,581 flip-flops which corresponds to 11% LUT usage and 5% FF usage of the FPGA deployed by AWS. Their design is much larger than [66] and [67] but they still only used a small portion of the device. These design are implemented on a different FPGA target than the AMD VU9P. Values in Table 2-1 may not be fully accurate due to hardware difference. Those values are still useful to understand the order of magnitude of FPGA-based accelerator's resource utilization.

Resource utilization of FPGA accelerators for AMD VU9P				
Related work	LUT	FF	DSP	BRAM
[65] Xiao et. al	131,042 (11%)	113,581 (5%)	242 (3.5%)	4 (5%)
[66] Tsai et. al	38,899 (3%)	40,534 (1.7%)	9 (0.1%)	3 (4%)
[67] Zhou et. al	80,175 (7%)	46,140 (2%)	83 (1.2%)	0
VU9P resources	1,182,000	2,364,000	6,840	75.9

Table 2-1 – Resource utilization of 3 FPGA-based neural network accelerators

Currently, FPGA cloud services lack of resource usage efficiency [68]. FPGA multi-tenancy is one solution to address this issue. It allows seamless sharing of FPGA devices among multiple users. There are two types of FPGA multi-tenancy: spatial multi-tenancy and temporal multi-tenancy. In spatial multi-tenancy, each user has a well defined set of FPGA resources called Partially Reconfigurable Region (PRR). As shown in Figure 2-11, users are able to deploy their acceleration solution in their PRR independently from other users. Each tenant can use his own PRR in any way at any time. This way FPGA usage is maximized and computing resources are not left unused. Due to its architecture, an FPGA device can run multiple acceleration functions in parallel.

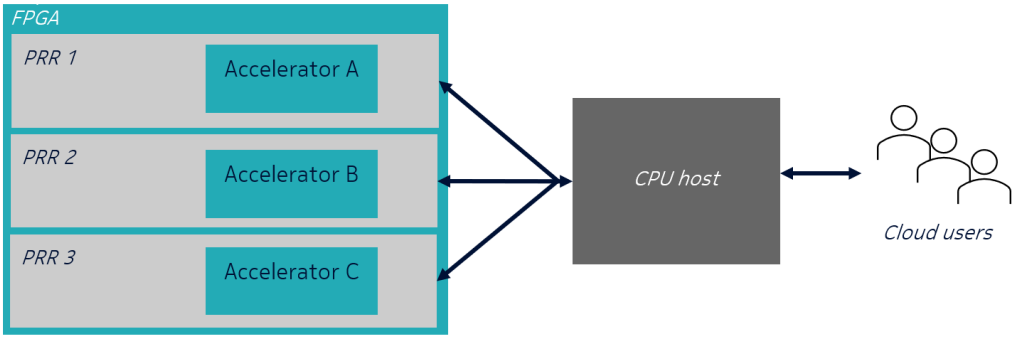


Figure 2-11 – Example of spatial multi-tenancy for FPGA

Temporal multi-tenancy (i.e., time multiplexing) is another method to have multiple tenants use the same FPGA. Each user has a time slot where he can use the FPGA. This technique does not maximize resource utilization but it allows multiple users to utilize a single FPGA. Temporal multi-tenancy is considered to offer better isolation compared to spatial multi-tenancy because users are not using FPGA re-

sources at the same time. This removes any possibility to exploit data leakages and side-channel vulnerabilities [69].

To the best of our knowledge, no cloud service provider supports multi-tenancy for FPGA. However, FPGA multi-tenancy is not a straightforward challenge. Due to the multi-user environment, access control and user isolation are prominent challenges [70], [71], [72], [73]. Some research in FPGA multi-tenancy aim to find an efficient solution to share the FPGA device among users [52], [71], [74], [75], [76]. FPGA virtualization is one crucial tool to achieve FPGA cloud multi-tenancy.

5.2 FPGA virtualization solutions for multi-tenant cloud computing

Virtualization is an essential tool to deploy FPGA devices in the cloud with multi-tenancy feature. It allows to create a virtual FPGA acceleration environment by using physical resources like logic cells, interfaces and memory available on the FPGA. Virtualization for FPGA clouds allows to deploy and manage smaller virtual FPGA instances. Table 2-1 shows that a high-end FPGA for cloud is underutilized with a single application/user. Typical accelerators for AI/ML like [65] only managed to use 11% of the available LUT resources.

There are two paradigms for FPGA virtualization. One of them is full virtualization. With this method, a hypervisor is used to abstract the physical hardware to create virtual resources. It is responsible for creating, monitoring and managing virtual FPGA instances using defined virtual resources [71], [75]. This component can be deployed with different architecture and can be hardware or software [77]. With full virtualization, Virtual Machines (VM) are deployed and virtual resources are allocated to them. Each deployed VM is running independently as if it has its own physical resource. Multiple independent operating systems and workloads can coexist on the same physical layer isolated from each other. VMware vSphere, Microsoft Hyper-V, KVM and Xen are well-known hypervisors that offer a range of features and capabilities, catering to different virtualization needs and enabling efficient management of virtual machines [78], [79], [80], [81].

Another paradigm of virtualization is containerization or OS-level virtualization. It is a newer form of virtualization that works at the operating system level. This method allows to deploy multiple lightweight containers to share the same OS-kernel. A container is a package of applications and their dependencies. Containers are considered more lightweight and faster compared to VMs since they do not require

separate guest OS for each instance. Containerization is popular for micro-service architectures and scalable cloud applications. Docker is the most widely used container runtime, but other solutions like Podman and Containerd also exist [82], [83], [84]. A container orchestration solution is deployed alongside a container runtime to benefit from large-scale container deployment. Such a solution allows to facilitate the management and the deployment of containers. A container orchestration can handle the lifecycle of containers and load balancing to distribute traffic efficiently. Security is another aspect that can be handled with a container orchestration solution. Mechanisms such as access control and network policies can be managed to safely deploy containerized applications. Other mechanisms like monitoring, scaling and seamless rolling updates can also exist depending on the solution. Kubernetes is the most popular container orchestration tool alongside Docker Swarm and Apache Mesos [85], [86], [87].

Many solutions are proposed in the literature to apply virtualization principles for multi-tenant FPGA cloud computing [68], [71], [72], [75], [88], [89], [90]. FPGA virtualization can be approached from multiple perspectives like the analysis of virtualization techniques, exploration of diverse use cases, and examination of various execution models. In [88], FPGA virtualization is classified in three levels. The first of them is resource level virtualization where a resource can be "configurable" or 'non-reconfigurable". At this level, architecture and I/O virtualization are considered and virtualization principles are applied closer to the FPGA hardware resource/architecture. FPGA overlays for architecture abstraction [91] and I/O sharing [92] are examples of FPGA virtualization solution at this level. In [68], a full-stack FPGA virtualization solution is proposed. The solution provides and abstraction of FPGA architecture to decouple compilation of FPGA code from FPGA-specific resources. Additionally, they provide virtualization for peripheral components like DRAM and ethernet with isolation support for multi-tenant environments. Each user gets a virtual machine with a virtual FPGA according to resources allocated to the user.

Node-level (i.e., FPGA level) virtualization represents the second level of virtualization introduced by [88]. In this level, higher-level mechanisms such as FPGA deployment, resource management, and orchestration are introduced. In this context, FPGA hypervisors, virtual machine support, FPGA shells (i.e., static hardware modules) and runtime systems (e.g., containers) are developed for FPGA cloud usage [68], [70], [71], [72], [77], [89]. For example, the solution proposed by [71] supports provisioning multi-tenant FPGAs in KVM clouds. The architecture proposes an approach based on virtual regions as opposed to an entire FPGA. In this work, authors introduced their own definition of multi-tenancy and elasticity within the

context of hardware/software virtualization capabilities. The KVM infrastructure is also extended to facilitate FPGA allocation and access within KVM clouds.

Cluster-level virtualization is the third level of FPGA virtualization. A cluster is a group of two or more FPGAs that are working together to accomplish the same task. Cluster-level virtualization defines an architecture to connect and manage FPGA clusters. Solutions like, project Brainwave and project Catapult [93], [8] use FPGA clusters to deploy a neural network model across multiple FPGA. The deployment architecture is described in Figure 2-2.

All deployment solutions presented in this chapter have security mechanisms to provide isolation and security at the software level. In full virtualization paradigm, VMs are providing isolation between tenants. With OS-level virtualization, user isolation can be provided by setting network policy for containers. The FPGA must also be secured at the hardware-level because users deploy hardware solutions in the FPGA fabric. These hardware solutions may have interfaces with external memory, shared IP modules, buses and more. In a multi-tenant context, it is utterly important to isolate and secure the hardware execution environment with mechanisms like authentication and access control. Existing cloud deployment solutions lack comprehensive support for FPGA deployment. Consequently, researchers are actively exploring and enhancing cutting-edge cloud technologies to incorporate FPGA-specific deployment tools and mechanisms.

5.3 Security vulnerabilities of cloud-based multi-tenant FPGA

5.3-1 Hardware-based attacks

Multi-tenant FPGA cloud environments have a wide range of vulnerabilities. However, high-level software vulnerabilities of hypervisors and other virtualization tools are not considered in this section. Instead, this section focuses on attacks that can be executed using FPGA logic. These attacks can be classified as side-channel attacks, fault injection attacks and denial-of-service. These security flaws are often created by the FPGA architecture. For example, the Power Distribution Network (i.e., PDN) inside the FPGA is one architectural vulnerability that creates a side-channel. These types of vulnerabilities cannot be patched or fixed after the device manufacturing process. The PDN is responsible of powering the FPGA. In multi-tenant use case, tenants share the same power distribution network. However, the switching activity generated by each tenant accelerator solution adds noise into the PDN. This leads to data leakage into the PDN. Multiple research show that by

collecting power traces using Time-to-Digital Converters (i.e., TDC) it is possible to retrieve another tenant's encryption keys [94], [95], [96]. A TDC is a logical circuit that can use latches or Ring Oscillators (i.e., RO) [69] to measure a signal's propagation delay. This allows to observe voltage fluctuations because delay and voltage are linked. As a consequence, the TDC is used as a delay sensor to monitor voltage fluctuation to collect power traces. By doing Correlation Power Analysis (i.e., CPA) it is possible to retrieve other tenant's data. To fix this issue, one can generate artificial switching activity to add noise into the PDN. An active fence is a type of power waster that is placed between the victim and the attacker. Switching activity can be generated with a RO [97] or wires [96]. However, active fences add power consumption and resource utilization overhead. In [97], the active fence consumed 178 µW to protect an AES module that required 320 µW. This results in a 56% increase in power consumption to protect the encryption key. Additionally, the active fence is using 678 LUTs for [97] and 2048 LUTs for [96]. The overhead for active fences vary according to its architecture, the partial reconfiguration region placement, and the module that needs to be protected. However, instead of measuring signal delay to get a read on voltage, one can create voltage fluctuations. This introduces additional delay to signals and it can be used to execute fault injection attacks.

Power wasters are also used to perform fault injection attacks. The variation in dynamic power consumption within FPGAs emerges from the switching capacitance (i.e., C) of logic signals which occurs at a frequency (i.e., f) when these signals transition between lower and higher voltage levels (i.e., V).

$$p_{dyn} = V^2 \cdot f \cdot C \tag{2-1}$$

For example, RO are used to introduce high frequency switching activity which increases power consumption. Figure 2-12 is an example of a RO using an AND gate and three inverters. The number of inverters must be odd to effectively oscillate. Because the PDN is shared in a multi-tenant context, the attacker aims to create voltage fluctuations with the goal to induce timing faults within the victim logic [98], [99]. However, due to its frequent use in malicious designs (e.g., delay sensor, power waster), RO are detected and prohibited by AWS [3], [100]. Prior to the deployment of FPGA logic, AWS runs multiple Design Rule Checks (i.e., DRC) on the cloud user's logic design to enforce cloud security. The cloud user must send an FPGA design netlist where all signal interconnections are verified. Prohibited design patterns (e.g., combinatorial loops) and connections are searched during this process. For example, the RO in Figure 2-12 is a combinatorial loop because it is not reached

by any clock signal and it forms a loop. Thus, this kind of FPGA logic cannot be deployed in AWS FPGA cloud. Any FPGA logic that needs to be deployed into an AWS cloud solution must successfully pass this verification step. Yet, researchers found a way to bypass the AWS DRC and implement malicious FPGA logic. In [11], various types of valid RO designs are detailed. FPGA primitives like MUX, CARRY, DSP, and FF are used to build a RO that can successfully bypass AWS DRC. As a consequence, [100] demonstrated the first DoS attack on AWS FPGA cloud instance. It is also possible to execute fault injection attacks with any type of RO. RAM-jam [101] uses the dual port RAM memory in FPGA to create short-circuits and collisions by writing concurrent opposite values into the memory. This leads to increased device temperature up to 110°C, voltage drop of 10% (maximum value) and bit-flips in the memory. As temperature and voltage drops have a negative impact on signal delay, RAM-jam is used to inject faults into FPGA logic. Authors successfully introduced fault into a deep neural network by creating bit-flips to neural network weights stored in memory. Nowadays, neural networks are used for critical applications like autonomous driving and classification algorithms for disease treatment. Such vulnerability can have tremendous consequences. Additionally, a Finite-State Machine (i.e., FSM) is also targeted and authors were able to create a transition into an unexpected state. They successfully attacked a password authentication FSM.

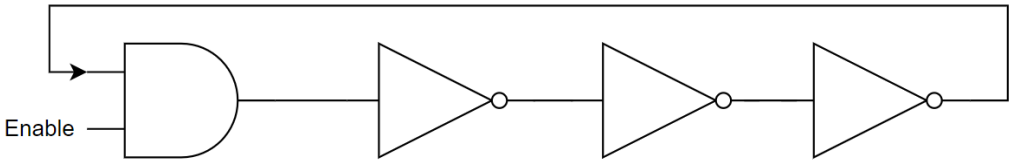


Figure 2-12 – Example of a ring oscillator using three inverters

5.3-2 Mitigation for hardware-based vulnerabilities

There are few ways to mitigate voltage and power attacks. One method is design verification. For instance, AWS, Huawei and Alibaba have FPGA design verification (i.e., DRC) for their cloud infrastructure [3], [5], [13], [102]. Unfortunately, their solutions are not open-source. Each CP verifies user FPGA design by analyzing signal connection patterns to find known malicious designs. However if a pattern is unknown to the verification tool, a malicious design can still be validated and implemented in a cloud solution. FPGA design verification is a recurring pattern of identification and evasion. Some solutions in the literature have successfully validated the design verification steps despite having implemented malicious circuits [100], [103], [104]. Other FPGA verification solutions exist in the literature. FPGADefender [11],

[100] is an open-source bitstream verification software that looks for malicious patterns like short-circuits, long wires, combinatorial loops (e.g., Figure 2-12) and more. FPGA bitstream format is not documented by FPGA vendors. Authors of [11] built their software thanks to a prior reverse engineering work on bitstreams. According to available information, FPGADefender is the first open-source bitstream verification tool. Proprietary and closed-source technologies of FPGAs (e.g., bitstream format, drivers, etc.) are slowing down the development and the adoption of such devices in industrial application [105]. This also limits the collaborative and open-source capabilities of this technological area.

Another method to mitigate previously mentioned attacks is runtime protection. This type of defense is not common practice due to its complexity. Protecting the FPGA fabric at runtime is challenging because FPGA do not have a runtime execution environment. Hence, runtime security solutions can be harsh on logical resource consumption. This is not suitable for multi-tenant FPGA clouds because it reduces the amount of logic that can be allocated to each tenant. Yet, the first runtime security solution for multi-tenant FPGA clouds is described in [106]. Loopbreaker allows to detect and stop voltage-based attacks at runtime. In this solution, a TDC is implemented to measure voltage fluctuations inside the device. When fluctuations cross a predetermined threshold, the malicious tenant is identified and its reconfigurable region is reset to a safe state. This is done with a blanking bitstream. It is an FPGA configuration file that has no logic. It is used to reduce power consumption when the deployed logic of a PRR is no longer needed. The FPGA vendor tool already provides a blanking bitstream. When Loopbreaker detects a voltage attack, the blanking bitstream is sent to the ICAP (i.e., Internal Configuration Access Port) to reconfigure the malicious user's logic region. Reconfiguration latency is critical because the voltage attack must be stopped swiftly before creating timing faults or FPGA crashes. Because bitstream reconfiguration latency is linear to bitstream size, authors of [106] built a lightweight blanking bitstream by reverse engineering the bitstream file and its commands. Their blanking bitstream is 128.2 % faster compared to the blanking bitstream generated by the FPGA vendor tool. As a result, Loopbreaker is able to stop more attacks compared to the naive approach with the vendor blanking bitstream. This solution can prevent 100 % of FPGA crashes due to latch-based attacks and between 60 % to 100 % of crashes due to RO-based attacks. One drawback of this study is the absence of an identification mechanism for the malicious tenant's PRR. Nevertheless, such a mechanism is described in work [107]. This solution is based on an array of voltage sensor to determine the location of the voltage attack. This is possible because logical regions closer to the attacker logic

are more affected than those farther away.

Previously described security flaws are coming from low level elements (e.g., architecture, device voltage). There are also higher level security vulnerabilities that can impact an FPGA cloud. Some of these flaws are access control and memory isolation. FPGA technology and solutions are constantly evolving. Hence, some mechanisms can become obsolete under few circumstances. This is the case with cloud and multi-tenant environments. Memory protection mechanisms like Xilinx Memory Protection Unit (i.e., XMPU) and Xilinx Peripheral Protection Unit (i.e., XPPU) exist for AMD FPGAs. These configurable module are hardwired inside the FPGA logic. They protect different portion of the device like external memory (i.e., DDR), IO ports and processors. However, this kind of protection is not enough for multi-tenant context. XMPU enforces a binary type of security: authorized or unauthorized to communicate. For multi-tenant FPGA, a fine-grained memory isolation mechanism is necessary to separate each tenant's memory. This way, tenants cannot interfere with each others data stored in memory. Work [108] brought a solution for this issue. A hardware extension of the XMPU is proposed where up to 16 distinct memory regions can be allocated to users. Like XMPU, an address-restricted scheme is used where each master address needs to be explicitly allowed to communicate with a specific slave address. This solution can be seen as an access control mechanism where users are isolate by their addresses. A user cannot make memory operations with a memory region he is not authorized to access. Similarly, work [109] addresses access control issues for multi-tenant FPGA use case. It allows to safely and securely share on-chip BRAM (i.e., Block RAM). This is achieved with a new architecture of an AXI crossbar IP including one orchestrator, a mapping table, and arbiters for masters and slaves. The orchestrator manages the run-time management for security configurations and arbiters provide security verification for AXI transactions.

The deployment framework of an FPGA cloud is of the utmost importance to benefit from a secure multi-tenant FPGA cloud acceleration. It is the backbone of multi-tenant FPGA cloud security because all mechanisms and methods (e.g., communications, isolation, reconfiguration) are defined with the cloud FPGA framework.

5.4 Multi-tenant FPGA architecture for cloud computing

In contrast to Section 4.3 where the focus was on the involvement of the TA, in this section we focus on the mechanisms and the architecture of the FPGA to provide a secure multi-tenant environment. There are important characteristics and

metrics to offer a secure multi-tenant FPGA architecture for accelerated cloud computing. It is crucial to recall that the FPGA fabric is the most valuable resource in FPGA cloud acceleration. From an FPGA cloud user standpoint, having access to large amount of logical resources is preferable. It offers implementation freedom and loosens design constraints. However it is the opposite for the cloud provider: more PRR means more users. Hence, resource consumption and cost effectiveness is maximized. A compromise between efficiency and security must be found for both parties. Another important aspect is the scalability of the multi-tenant architecture. To enforce security in the FPGA, logic resources must be used. Depending on the architecture and the requested security level, the computational overhead and the resource consumption can vary. If the deployed solution introduces too much latency, the concept of "FPGA cloud acceleration" looses its meaning. However, if the deployed architecture addresses very few vulnerabilities, it could be harmful for the cloud infrastructure and user security [100], [103], [104], [110].

In [52], an FPGA cloud architecture is described. This solution uses a TA to create isolation between the user and the CP. The advantage of involving a TA in the FPGA cloud architecture is discussed in Section 4.3. In the proposed architecture, each FPGA has a certificate and it is included inside a PKI. To secure the acceleration environment, a device master key is generated by a PUF. The master key is unique and it is under the control of the attestation root-of-trust (i.e., TA). The master key is derived using deterministic key derivation functions to generate additional keys for various purposes like attestation keys, bitstream encryption keys and enclave keys. This architecture is multi-tenant. Each tenant is placed in a PRR called enclave. The latter is secured with public key cryptography and a certificate to protect the user region and create isolation. One drawback of this architecture is the extensive use of public key cryptography which can lead to increased latency and resource consumption. Another notable limitation is the device master key generated by a PUF. The master key can be exposed to the device owner (i.e., cloud provider) because he has means to characterize the device PUF with various attacks documented in Section 4.2. If the CP can get the device master key, he can get all other keys obtained by deterministic derivation method. As a consequence, the CP can get full access to all the keys used in the device. This can cause bitstream and user data data theft. Additionally, the TA is managing the device master key which is a critical issue for device security. In work [52], the TA is fully trusted. If we consider a stricter threat model, the level of exposure to the TA may be suboptimal. Thus, it is important to define the capabilities of the TA and the risks associated to him. The key management may be another limitation for two reasons. Firstly,

there are more than three symmetrical keys, five pairs of asymmetric keys and one pair of asymmetric key per tenant for one FPGA device. If we consider a large scale FPGA cloud deployment, managing these keys and certificates may be computationally costly and ineffective. The scalability of this FPGA cloud architecture is not ideal. Secondly, there are more than five keys that are stored inside the FPGA. Storing keys is not considered secure for FPGA clouds due to existing attacks vectors described in the previous section.

MeetGo [111] is a multi-tenant cloud FPGA architecture that provides a trusted execution environment for remote applications. To provide secure acceleration, the FPGA chip manufacturer generates an asymmetric key pair for the FPGA. The private key is embedded inside the FPGA and it is solely accessible by a hardware security agent. The latter is a hardware module that stores session keys and bitstream signatures. The ICAP is managed by the security agent to reconfigure FPGA regions. To establish a secure channel, the remote user gets the FPGA public key and uses it to negotiate a session key with the FPGA. This key is used to securely communicate (e.g., send a bitstream) with the FPGA. Before FPGA reconfiguration, the security agent decrypts the user bitstream and verifies its signature. The bitstream must be signed by a known entity (e.g., trusted third party) to be valid. The authors claim that the cloud user can securely embed encryption keys (e.g., symmetrical or private keys) inside their FPGA logic because the bitstream is sent encrypted. Embedding encryption keys into bitstreams in not a secure practice for FPGA cloud computing. Cloud providers require user bitstream verification as discussed in Section 5.3. This architecture is multi-tenant. Each user has a PRR for his own user application. Users are logically isolated from each other and no component like memory or interfaces are shared between tenants. While this is great for security, it is at the detriment of efficiency and flexibility for the FPGA acceleration environment.

In [112], authors developed and analyzed an FPGA resource pooling system. Their solution allows a user to use FPGA acceleration resources in a cloud environment with a high level of abstraction. Users can bring their own bitstreams or can use the designs provided to meet their acceleration needs. Their paper is heavily focused on FPGA cloud efficiency and performance. The offered performance improvement is great, but nothing is mentioned security wise. There are few potential security threats with the proposed scheme. If the user brings his own design, his IP is on the one hand totally disclosed to the CP. The user has no confidentiality and this is unsuitable for the user. On the other hand, no design integrity mechanisms are mentioned in the paper to ensure that the user IP is delivered without being modified

or compromised. And lastly, the paper does not mention any mechanism to protect the FPGA device from the user design. Nowadays, we know that circuits like power waster can be very harmful to FPGA devices and cause great economical damage. Another security aspect that is not mentioned is authentication. The user never authenticates the FPGA and has no proof that the allocated FPGA is the one he used. This can lead to FPGA impersonation and the FPGA can be compromised. This can cause data breaches and malicious behaviors. The fact that the user is also not authenticated by the FPGA reinforces those security flaws. The FPGA, or a reconfigurable region inside a device as stated in the article, lacks of isolation. A user may be capable to access a device allocated to another user if no user-FPGA authentication is done.

Authors of [7] have developed a shielded enclave for FPGA cloud accelerators. As shown in Figure 2-13, the CP, the data owner (i.e., DO) and the IP vendor are taking place in this architecture. The data owner needs acceleration and uses the IP vendor accelerator design. The IP vendor protects the accelerator requested by the data owner by wrapping it with a shield IP. Under this scheme, an asymmetric shield encryption key is generated by the IP vendor. The private key is embedded inside the shield and the public key is shared with the data owner. To create an encrypted channel with the FPGA, a symmetric data encryption key is generated by the data owner. Then, the data encryption key is encrypted with the shield public key and sent to the FPGA by the data owner. The encrypted data key (i.e., load key) is decrypted inside the shield using the embedded private key. As a result, the shield and the data owner have a shared symmetrical data encryption key (e.g., AES encryption key) for future data transfers. Data integrity is also ensured with digest algorithms (e.g., HMAC). Data protection and privacy problems in public cloud environments are addressed by placing a shield IP between the IP vendor's accelerator logic and the FPGA interfaces provided by the CP. Any data going out of the accelerator is encrypted and any data going inside the accelerator is decrypted using the data encryption key with an AES block. The solution described in [7] also includes a security kernel and a security processor block for attestation, bitstream configuration and secure boot mechanisms. The final FPGA binary (i.e., bitstream) containing the accelerator, the shield IP and the embedded private shield encryption key is encrypted with a bitstream encryption key generated by the IP owner. It is possible to use multiple shielded enclaves with different users which allows spatial FPGA multi-tenancy.

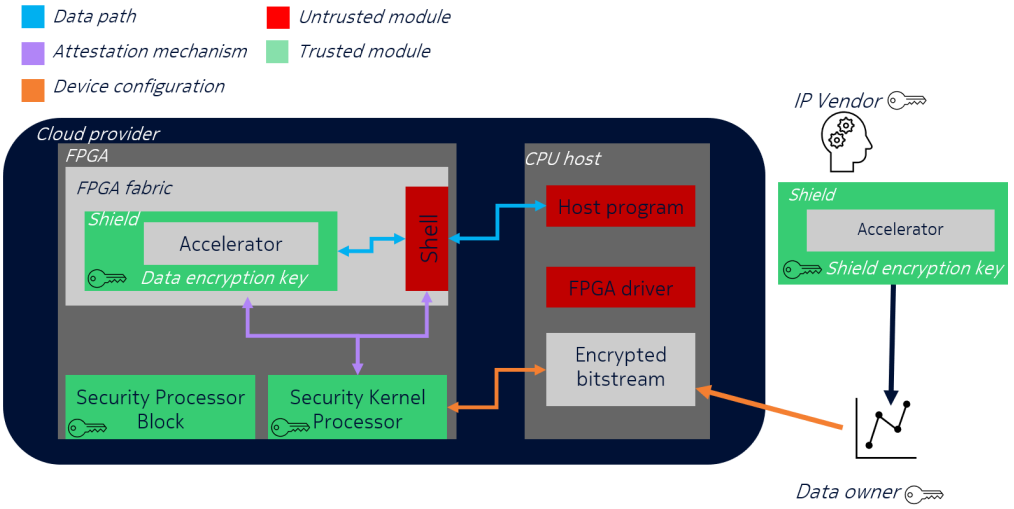


Figure 2-13 – ShEF shielded enclave architecture

Table 2-2 summarizes the contributions of each work. All the presented architectures successfully authenticate the FPGA by involving a TA in the authentication scheme. However, the cloud user is never authenticated in any of the works. In fact, the user authenticates the FPGA according to a scheme and sets up a session keys with afterwards. This means that the necessary credentials for FPGA authentication and session key setup is critical for the remote computation security. An attacker can set up a key with the FPGA if he gets the required credentials to complete the scheme. At the beginning of Section 5.4, the importance of efficiency and multi-tenancy has been underlined. Work [53] and [63] do not support multi tenancy while being scalable on a larger deployment environment and the overhead that comes with the solution is acceptable. Work [52] and [111] do support multi-tenancy. However, [52] heavily relies on multiple keys, key derivation and multiple certificates per FPGA. Security-wise, this increases the attack surface due to multiple keys being deployed and used. Besides, deploying multiple certificates per FPGA (e.g., boot signature, bitstream signature...) may impact the scalability of the architecture on a larger scale. Work [111] has a better scalability. The FPGA only has an asymmetric key and one symmetric session key per PRR. However from the security perspective, the security agent knows each user session key. Thus, user data can be retrieved in plaintext by the security agent. The user cannot verify the version or the legitimacy of the security agent. Because the latter is responsible of the data encryption using the session key, it can also collect user data. Hence, user and CP isolation may not be optimal. Work [52], [53] and [63] protect the user against the CP but they are more exposed to the TA. The concept of a trusted third party (i.e., TA) is sometimes used to provide isolation between two stakeholders. By definition a TA is trusted.

Yet, it is not acceptable to have too much exposure to a TA. This means not giving full device control and capability to the TA. For example in [52], the TA controls the device master key, and all certificates. As a result, the TA takes the job of the CP. The latter becomes infrastructure and device owner and the TA turns into a security manager role. If the amount of exposure in an architecture with a TA is similar to the exposure in an architecture without a TA, no additional security and isolation is provided. In fact, the exposure has only been moved from the CP to the TA.

Comparison of multi-tenant cloud FPGA architectures					
Achievements	[52]	[111]	[7]	[112]	This work
FPGA Authentication	✓	✓	×*	×	✓
User Authentication	×	×	×*	×	✓
Multi-tenancy	✓	✓	✓	✓	✓
Scalability	×	✓	✓	✓	✓
User-CP isolation	✓	×	✓	×	✓
Exposure to TA	+++	+	+	*no TA*	+
Access control	×	×	×	×	✓
Zero-trust secure comms.	×	×	×	×	✓
Manage & share access	×	×	×	×	✓

×* Only uses authenticated encryption algorithm, no actual authentication

"+++" means more exposure to TA than "+"

Table 2-2 – Comparison with prior works on multi-tenant cloud FPGA architecture

The architecture presented in Chapter III uses a TA and a CP in a more balanced way to split responsibilities and render them incapable of acting independently of each other. This way, the architecture's security benefits from the presence of a TA without being too much exposed to a single entity. From an efficiency aspect, the proposed architecture allows multiple users to use the same memory interface securely and seamlessly. Thanks to a hardware security module (i.e., HSM) that en-

forces access control and address translation mechanisms, each user's memory region is isolated from each other. The HSM randomly allocates memory regions to each tenant inside the FPGA, this allows to mitigate memory attacks like Rowhammer. The HSM can filter memory transactions coming from user regions and determine where the data needs to be written/read. As said earlier in this chapter, embedding encryption keys inside the user bitstream is not a secure practice because cloud providers have a mandatory bitstream verification procedure. The encryption can be accessed in plaintext by the CP and lead to data theft. In this work, a patented zero-trust secure channel establishment protocol to provide a secure communication channel between the cloud user and the FPGA (e.g., FPGA shell and user design) is proposed. This protocol does not require to embed sensitive encryption keys inside the bitstream and it is totally isolated from the CP. This protocol in executed in the HSM. This protocol is further described in Chapter III.

Additionally, the framework proposed in this manuscript uses an upgraded hardware shield enclave to protect the user accelerator. Initially, the authors proposed the shield module to protect the intellectual property of an IP vendor against the cloud user and the CP. While the IP is protected against the user, it is not the case for the bitstream verification procedure of the CP. With the use case proposed in [7], the user encryption key is exposed to both the CP and the IP vendor. In fact, the user must generate a symmetrical encryption key and encrypt it with the IP owner's public key. The IP owner then includes the encrypted key inside the bitstream. However, the IP owner has the private key associated to the public key. He has the capability to decrypt the cloud user's data if he intercepts communications between the cloud user and the FPGA. The current work proposes an upgrade to the existing shield architecture by adding the zero-trust secure channel establishment protocol. This extra security layer helps to isolate and offer better data protection compared to the literature.

In this section, the lack of efficiency in single-tenant FPGA and the advantages of multi-tenancy is discussed. Further detail is given on technologies like virtualization which enables FPGA multi-tenancy. Then security vulnerabilities of multi-tenant systems are exposed and existing architectures are presented. However, multi-tenancy is an emerging concept in the realm of FPGA cloud computing. There is a need for enhanced security inside the FPGA because multiple users are sharing the same hardware. The following section describes access control mechanisms for multi-user FPGA acceleration environments. Access control is an essential security measure to isolate users from each other while mediating shared resource access and usage.

6 Access control mechanisms for multi-tenant FPGA clouds

Access control is an important security tool to enforce user authorization rules set by the access manager. The cloud provider often bears the responsibility to give authorizations users in order to limit their capabilities. A user must be authenticated and identified to obtain authorization from a provider. Additionally, user identification is also necessary to enforce access control rules. Multi-tenancy is not a widespread mechanism in current deployments and to the best of our knowledge, no cloud provider proposes multi-tenant FPGA. Access control is a well studied subject for conventional systems but it is relatively new for cloud-based FPGAs. Due to their inherent nature, FPGA cannot naively use the state-of-the-art access control techniques developed for conventional systems. Access control mechanisms must be extended or specifically developed for FPGA systems.

In that regard, an AXI-based access control mechanism for multi-tenant FPGA is proposed in [113]. An AXI crossbar module manages the communication between master and slave IPs. Authors propose to upgrade this module by adding access control, scheduling mechanisms to meet security and efficiency requirements for multi-tenant FPGA environments. This upgraded module uses mapping tables to securely share resources between master IPs and arbiters to enforce rules. Additionally, an orchestrator can dynamically update access control rules enforced inside the AXI crossbar by adding or removing authorizations given to master IPs. Arbiters located inside the upgraded AXI crossbar module check the protocol validity, address violations and provide efficient resource sharing using mapping tables and round-robin scheduling. This solution allows to safely share slave IPs with masters by enforcing dynamic access control rules.

Work [114] is a higher level solution compared to the precedent work. It uses virtualization concepts in a multi-tenant FPGA cloud to provide isolation and efficiency. In this work, a hypervisor named Optimus is described. It aims to provide virtualization for shared FPGA memory. To ensure isolation between tenants, a hardware-based virtualization control unit configures virtual memory and IO address offsets located inside a hardware monitor module in the FPGA PL. To interact with FPGA DRAM, tenants use virtual addresses. The hardware monitor translates the guest virtual addresses (used by tenants) to IO virtual addresses and sends the requests to the host CPU. The Memory Management Unit (i.e., MMU) translates the IO virtual address into physical address using a mapping table. To ensure tenant

memory isolation a page table slicing technique is adapted to FPGA. This directly maps IO virtual addresses to guest virtual addresses in a contiguous way. This technique creates isolation between guest memory accesses and provides efficient DMA addressing because each memory partition is assigned to a specific guest accelerator. In this work, isolation and access control are achieved through virtualization and it is seamless for the FPGA user. There are similar solution leveraging virtualization techniques to isolate users exposed in Section 5.2 and in [115], [116], [117]. They utilize virtualization methods to securely deploy FPGA accelerators in a multi-tenant execution environment. However, these works do not address security issues located inside the PL (e.g., hardware isolation, side-channel).

7 Summary

This chapter provides a comprehensive summary of the realm of multi-tenant FPGA cloud computing and its associated security requirements. Current cloud-based FPGA solutions and services are introduced in Section 1. The FPGA cloud architectures of IBM and Intel (e.g., Brainwave and Catapult) are described. Additionally, prominent cloud service providers such as Amazon, Alibaba, and Huawei are listed, and their respective requirements are discussed. These requirements include bitstream verification and the prohibition of certain logic design patterns (e.g., combinatorial loops). In Section 2, Trusted Execution Environment (i.e., TEE) environments for FPGA are described. Technologies like Arm TrustZone, Intel SGX, and Keystone are detailed. While each TEE employs a distinct method to create a secure execution environment, their common goal is to achieve isolated and secure computing in an untrusted environment. In the context of FPGA cloud computing, users seek isolation from the Central Processor (i.e., CP). Thus, TEE solutions represent one possibility to achieve secure software execution. To enhance security further, Section 3 discusses existing authentication techniques. Algorithms such as HMAC and Authenticated Encryption (i.e., AE) solutions like AES-GCM allow for message encryption while ensuring message integrity and authenticity. Unlike solutions such as Public Key Infrastructure (i.e., PKI), which rely on certificates and signatures issued by a trusted entity, these algorithms do not need a third party. Subsequently, in Section 4, FPGA-specific use cases are explored for bitstream and FPGA authentication. Additional FPGA-specific authentication methods using a Physical Unclonable Function (i.e., PUF) are described. The Section 5 highlights the shortcomings of single-tenant FPGA and describes FPGA multi-tenancy. The

concept of virtualization and its application to FPGA clouds are detailed, wherein virtualization tools establish a multi-tenant environment with isolation mechanisms. The section also identifies hardware-based multi-tenant vulnerabilities and proposes some mitigation techniques. The last section provides insights into FPGA access control within multi-tenant FPGA cloud environments. It is a crucial security element for securely sharing resources among FPGA tenants.

The architecture described in the next chapter is a token-based FPGA cloud access scheme based on OAuth 2. The latter has been adapted to share FPGA located in a cloud environment using access tokens. As shown in Table 2-2, this solution allows an authorized user to create and manage additional child tokens based on his access token thanks to a TA. This framework enables flexible, efficient and, secure remote FPGA acceleration compared to the literature.

Chapter III

Token-based multi-tenant FPGA cloud security

This chapter proposes a framework called TokSek. It is a token-based multi-tenant FPGA cloud security framework. To the best of our knowledge, it is the first framework to introduce the concept of access tokens to the FPGA realm. TokSek is an adaptation of OAuth 2 [55] to share access to FPGA devices instead of credentials or applications. This chapter proposes a modelization of the framework in Section 1 by establishing a threat model and providing a mathematical formalization of the mechanisms available in TokSek. Section 2 provides an introduction to TokSek and its capabilities. It is a software implementation of a token endpoint on FPGA. Additionally, a user to user access delegation mechanism is described and a real world use case is detailed. The framework is implemented and analyzed in Section 3 and experimental results are exposed.

1 Modelization of the framework

1.1 Threat model

The proposal considers the following threat model. One of the attacker's main objective is to steal valuable Intellectual Property (i.e., IP) and data from FPGA cloud users. Hence, the CP is considered as a potential threat because it can access user data in current FPGA clouds. In this work, threats coming from malicious tenant IP are not taken into account. In this attack family, attacker IPs are instantiated to exploit side-channels and retrieve sensitive information. This attack vector can be avoided with bitstream analysis. Hence malicious IPs are not a part of the threat model. Moreover, the embedded Linux operating system described in Section 3 is assumed to be secure. Attacks vectors on this technical solution are not considered. In this work, the threat model is considered in two parts. First, threats outside the

FPGA are detailed. Then the threats inside the FPGA are analyzed.

1.1-1 Threats outside the FPGA

The security of the protocol used to get remote FPGA access should not be neglected. First we define the threat model outside the FPGA. Figure 3-1 illustrates the threats coming from the outside of the FPGA cloud.

User impersonations: An attacker can impersonate a user by stealing credentials and behaving like the victim user. The attacker tries to access the victims' resources and take action according to the victims authorizations. This threat can be challenging to detect, especially in the case of Man in the Middle (MitM) attack or replay attack [58].

Data integrity: At any time, the attacker can steal the user bitstream to add hardware Trojans [12]. User bitstream can be modified to add malicious components to slow down, crash or get valuable information from the user design. The attacker can also modify messages and payloads to provoke unexpected behavior and faults in the victims computation.

Confidentiality: The attacker can intercept every communications in this protocol. The attacker wants to obtain valuable data, IP and other information that can give access to the allocated cloud resource.

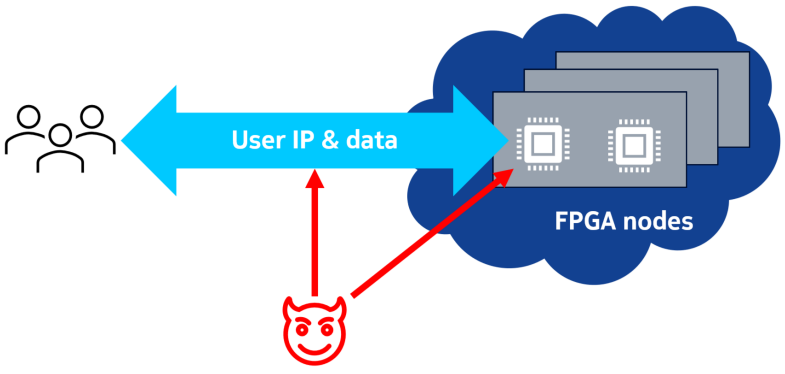


Figure 3-1 – Outside FPGA threat

1.1-2 Threats inside the FPGA

Once a secure access to the FPGA is established, the computation environment

inside the FPGA must be secured too. Figure 3-2 shows the possible threats coming inside the FPGA cloud.

Resource allocation: We assume that the attacker can choose and predict which FPGA is being attributed. The attacker also has prior information on the whereabouts of other tenants. The attacker can deploy fingerprinting methods to identify the FPGA cloud architecture as described in [12].

Logical isolation and access control: User isolation and access control in FPGAs is an important challenge. Due to the lack of a runtime environment (e.g., operating system) for the FPGA logic, addressing these challenges is not straightforward. Especially in a multi-tenant context, the isolation and access control requirements are higher. The attacker is able to exploit these weaknesses in order to gain access to unauthorized resources like user memory and cache content [73].

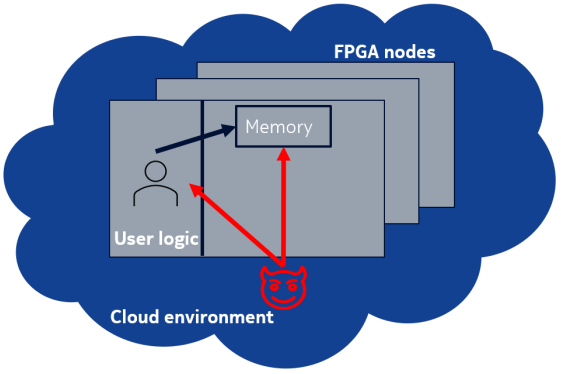


Figure 3-2 – Outside FPGA threat

1.2 TokSek modelization

The proposed framework model follows the principles described below.

— **Entities**

The proposed authorization and access delegation framework for FPGA clouds is running with four entities.

The CP is the owner and the manager of cloud resources. Let R be the set of resources deployed by the CP:

$$\forall j \in \mathbb{N} / r_0, ..., r_j \in R \tag{3-1}$$

Let U be the set of all users that benefit from the FPGA cloud:

$$\forall i \in \mathbb{N} / u_0, ..., u_i \in U \tag{3-2}$$

In this framework, the CP collaborates with the TA to share resources R with users U. Each entity in this framework is identified by certificates. Finally, each element of U and R have their own unique certificate.

— **Property 1**

Every user u_i needs a signed certificate from the CP to request resources R with the CP and receive access tokens from the TA. Let $cert\_sign$ be a function that returns a signed certificate to the user such as:

$$cert\_sign : cert(u_i, priv\_key(CP)) \longrightarrow signed\_cert(u_i) \tag{3-3}$$

— All communications of users U with entities defined above are mutually authenticated. Every entity checks the signed certificate $signed\_cert(u_i)$ to verify that it is signed by the CP. Users with invalid signatures are not able to communicate with entities present in this framework.
— Each token t_i is bound to a unique signed user certificate $signed\_cert(u_i)$. The public key of the user U_i must be inside the token t_i generated for him. Hence, $t_i \leftrightarrow signed\_cert(u_i)$.

— **Property 2**

Each resource r_j is associated to a secret and unique key FPGA Shared Secret (i.e., FSS) fss_j such as:

$$\forall i, j \in \mathbb{N} / r_i \neq r_j \Leftrightarrow fss_i \neq fss_j \tag{3-4}$$

Each resource r_j has its own Hardware Security Module (i.e., HSM). Moreover, each element of r_j is divided into multiple independent and reconfigurable regions $rr_{k,j}$. Using Equation 3-4 and the previous statement, R can be redefined such as:

$$R = \{r_j |\ r_j = \langle \{rr_{0,j}, ..., rr_{k,j}\}, hsm_j, fss_j \rangle j, k \in \mathbb{N}\} \tag{3-5}$$

— FSS and hsm_j are under TA responsibility. Each hsm_j has access to a unique fss_j that is stored inside a hardware memory enclave located inside the logic r_j. These are further described in Chapter IV.
— each fss_j is generated by a TRNG inside r_j independently from the CP. The TRNG is controlled by hsm_j.

Consequently, we can define the embedded resources er_j possessed by r_j such as:

$$\forall r_j \in R, \forall j,k \in \mathbb{N}\ \exists er_j \in r_j \mid er_j = \{memory, \{shrd\_ip_0, ..., shrd\_ip_k\}, \{rr_{0,j}, ..., rr_{k,j}\}\} \tag{3-6}$$

Inside the FPGA, resources like memory, reconfigurable regions and shared IPs or blocks can be allocated to users u_i.

— **Property 3**

This framework is token-based. Let T be the set of access tokens created by the TA:

$$\forall i \in \mathbb{N}/t_0, ..., t_i \in T \tag{3-7}$$

Each token t_i has a set of properties defined during their construction by the TA such as:

$$t_i = \langle header, payload, signature\rangle |\ payload = \langle signed\_cert(u_i), ID(r_j), Perm(u_i)\rangle, \\ signature = HMAC(payload, fss_j) \tag{3-8}$$

$Perm(u_i)$ represents the permissions given to user u_i and it is defined such as:

$$\forall u_i \in U, \forall j,n,p,q \in \mathbb{N}, \forall k \in [0,n], Perm(u_i) = \{er_{i,j,0}, ..., er_{i,j,k}\} \mid \\ er_{i,j,k} = \langle vld\_until_k,\ ID\{rr_{p,j}, ..., rr_{q,j}\},\ mem\_size_k,\ shrd\_ip_k,\ shrd\_mem\_size_k\rangle, \tag{3-9}$$

Equation 3-9 defines the permissions of u_i as a set containing permissions for embedded resources $er_{i,j,k}$. These permissions can be but are not limited to: token validity period, memory size, authorization to use shared hardware IPs and shared memory. It is important to notice that the set of reconfigurable region $rr_{p,j}, ..., rr_{q,j}$ allocated to t_i is not necessarily continuous.

— When creating a token t_i, the TA signs it with fss_j that is also known by resources R.
— $ID(r_j)$ identifies the resource r_j and $ID(rr_{k,j})$ its associated reconfigurable region.
— $ID(rr_{k,j})$ represents the targeted reconfigurable region of r_j and $Perm(u_i)$ describes the permissions given to u_i. It is used inside a resource r_j for the access control applicable to $ID(rr_{k,j})$.

As a consequence of Equation 3-8 and Equation 3-9, each token T gives exclusive

access to reconfigurable regions $rr_{k,j}$ such as:

$$\forall rr_{k,j} \in r_j, \forall u_i, u_j \in U, \forall t_i, t_j \in T, \forall V \in Perm(u_i) \; | rr_{k,j} \in V, \forall V' \in Perm(u_j), \\ t_i \neq t_j, \exists vld\_until \in V, \exists vld\_until' \in V' \implies \nexists rr_{k,j} \in V' | \; vld\_until = vld\_until' \tag{3-10}$$

The TA must keep track of valid tokens and prevent the CP from attributing allocated reconfigurable regions $rr_{k,j}$ to user tokens. Each reconfigurable region can be allocated to a single token.

— **Property 4**

Tokens T and users U can be authenticated. Let the function *auth_user* and *auth_token* such as:

$$\forall t_i \in T, \forall \; r_j \in R \; / \\ auth\_token(t_i, r_j) = true \iff \exists fss_j \in r_j | \; t_i(signature) = HMAC(t_i(payload), fss_j) \tag{3-11}$$

$$\forall u \in U, \exists \; signed\_cert(u) \; | \\ auth\_user(signed\_cert(u)) = true \iff cert\_sign(cert(u)) = signed\_cert(u) \tag{3-12}$$

The function *auth_user* is used by R, TA and CP to authenticate the users U for secure communication, resource request and access. In this chapter, the *auth_token* function is implemented in the PS. However, *auth_token* is implemented inside the HSM later in Chapter IV.

— **Theorem 1**

Users U can utilize resources R with the *access* function defined such as:

$$\begin{aligned} &\forall u_i \in U, \forall t_i \in T, \forall r_j \in R \\ &access(u_i, t_i, r_j) = true \iff auth\_user(signed\_cert(u_i)) = true \\ &\wedge \, auth\_token(t_i, fss_j) = true \wedge signed\_cert(u_i) \in t_i \\ &\wedge \, ID(r_j) \in t_i \end{aligned} \tag{3-13}$$

Only the authenticated user u_i can access the resource r_j associated to the access token t_i. The latter is associated to $signed\_cert(u_i)$ and signed with fss_j by the TA. This theorem ensures strong authentication of the user and its access token.

Attacker's perspective

— **Property 5**

Let Eve be an external attacker (e.g., not the CP) that tries to use a stolen token

or a stolen certificate such as:

$$\forall u_i \in U, \forall r_j \in R, \forall t_i \in T, t_{eve} = t_i, signed\_cert(u_{eve}) \neq signed\_cert(u_i) \vee$$
$$t_{eve} \neq t_i, signed\_cert(u_{eve}) = signed\_cert(u_i) \Leftrightarrow access(u_{eve}, t_{eve}, r_j) = false \tag{3-14}$$

Eve cannot steal and use tokens T thanks to Property 1. Nonetheless, let's consider that an attacker has successfully stolen a token t_i regardless of the method. Fourth property ensures that a stolen token t_i cannot be utilized by a malicious entity. In fact, each token t_i is associated to a user certificate $signed\_cert(u_i)$.

Eve can also steal the user certificate (both the public and private key) to access the user resource. However, Eve also needs token t_i to access the resources. Thanks to a strong authentication proposed in this framework, Eve needs a valid token and the associated user certificate.

— **Property 6**

Let's consider an internal attacker named Mallory (e.g., the CP) that has an interest in the data of u_i. Mallory tries to access or modify the content of token t_i such as:

$$\forall u_i \in U, \forall t_i \in T, \forall r \in R \exists\ t_i(payload_{mallory}) \in R \mid t_i(payload_{mallory}) =$$
$$\{signed\_cert(u_i), ID(r_{mallory,k}), Perm_{mallory}(u_i)\} \wedge auth\_token(t, fss_{mallory}) = false$$
$$\Longleftrightarrow signature = HMAC(payload_{mallory}, fss) \neq HMAC(payload, fss) \tag{3-15}$$

Mallory can in fact modify the access token t_i considering that she successfully stole it despite the mutually authenticated TLS channel. However, the access token can no longer be authenticated by the HSM because the token signature will not match the calculated signature.

Mallory can compromise the CP to allocate the already allocated reconfigurable regions $rr_{k,j}$ such as:

$$\forall u_i \in U, \forall t_i, t_j \in T, \exists rr_{k,j} \in r_j \mid rr_{k,j} \in t_i \wedge rr_{k,j} \in t_j \tag{3-16}$$

The attack modelized by Equation 3-16 cannot be executed thanks to Equation 3-10 in Property 3. In fact, the TA keeps track of attributed reconfigurable regions $rr_{k,j}$ and prevents the CP from giving access to busy reconfigurable regions.

Additionally, a compromised CP cannot generate permissions that the user did

not request. The attack can be described such as:

$$\begin{aligned} &\forall u_i \in U, \forall t_i \in T, \exists Perm_{requested}(u_i) \notin t_i, Perm_{obtained}(u_i) \in t_u \mid \\ &Perm_{obtained}(u_i) \neq Perm_{requested}(u_i) \end{aligned} \tag{3-17}$$

The user u_i has access to the content of access token t_i and can verify the permissions accorded to him. If u_i finds that Equation 3-17 is true, the token t_i can be rejected and not used.

This section provided a modelization for the framework described in the next section. The model formalizes the interactions between entities and provides a proof for security features like the access token and authentication methods.

The next section introduces TokSek: a token-based framework for secure multi-tenant cloud FPGA. The token-based authentication and authorization framework OAuth 2 is described in Chapter II Section 3.2. The capabilities of OAuth 2 and the high level interactions between different entities are detailed. OAuth 2 has been adapted to cloud FPGA usage for token-based FPGA access.

2 Introduction to TokSek

2.1 Authorization and token infrastructure

2.1-1 OAuth 2 adaptation to FPGA cloud

There are four major grant types in OAuth 2. One of them is the implicit grant that is more suitable for client-side applications where maintaining a client secret is challenging. Instead of issuing an authorization code for an access token, the latter is issued directly after the user authentication. However for a cloud FPGA usage, the implicit grant is not the most secure due to the lack of authorization code and missing extra authentication step. There are also more specific grants like the Resource Owner Password Credentials grant (i.e., ROPC) where the trusted clients handle sensitive user credentials. Its use is discouraged due to the lack of security and the requirement of trust. The third type of grant is the client credential grant where an access token can be obtained from an authorization server without involving a resource owner. This is not suitable for FPGA cloud because the CP must define the content of the access token and manage the resource utilization across its infrastructure. The fourth type of grant is the authorization code grant,

it is used in TokSek.

The authorization code grant flow in OAuth 2 is a robust and secure method, particularly adapted for applications where maintaining the confidentiality of the client secret is crucial for security. This flow begins with the user initiating an authorization request from the resource owner. Upon successful authentication, the authorization server issues an authorization code to the client. The critical security advantage lies in the separation of the front-end and back-end components. Unlike some other flows, the authorization code flow ensures that the sensitive client secret is never exposed or stored on the client side (e.g., browser or client device). Instead, the client securely exchanges the authorization code for an access token. This exchange happens with the back-end components of the authorization server and prevents to store any secret (e.g., user credential). This separation significantly mitigates the risk of unauthorized access and token interception, making it a preferred choice in scenarios where security is important. The use of the authorization code acts as a single use, short-lived token that provides an additional layer of security during the exchange process. If intercepted, it is of limited utility without the corresponding client credentials, hence reinforcing authentication requirements.

Often, OAuth 2 is used to share HTTP resources like user credentials. The OAuth 2 protocol must be adapted for the cloud use case for several reasons. Figure 3-3 shows the standard OAuth 2 flow and the modifications made for the TokSek flow. In a standard web-application, the resource owner (website user) uses the client's website (e.g., a blog) and is willing to share information. The client then asks for authorization for resource access (user data) and starts the access delegation procedure. The resource owner is using a service and wants to give access to his resources. This scheme is not valid in an FPGA-accelerated cloud use case because it is the user that requests access to the hardware owned by the RO. In a cloud FPGA case, the resource owner does not use a service but rather proposes a service for other users.

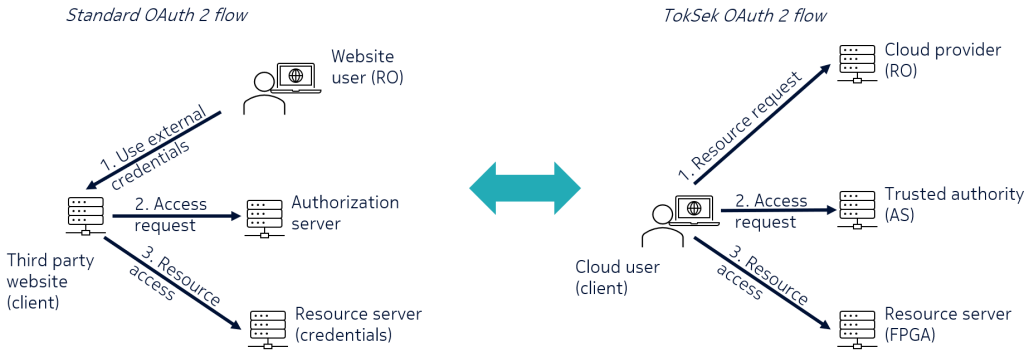


Figure 3-3 – Difference between standard OAuth 2 and TokSek OAuth 2

In an FPGA cloud context, the cloud provider is considered to be the resource owner, the trusted authority is the authorization server and the FPGA is a part of the resource server. The client requests a hardware resource with its account information. This is the first step of authentication. Then, resources are allocated automatically with the CP's cloud deployment tools (e.g., orchestrator/scheduler).

Access tokens are a core part of TokSek. It is a tool to represents claims and access cloud resources. It is also an important element to enforce access control rules inside the FPGA.

2.1-2 JSON Web Tokens for FPGA resource sharing

The first version of TokSek aims to bring token mechanisms into the cloud FPGA realm with a software approach. The infrastructure to use access tokens (e.g., server, libraries, host software) is implemented on the FPGA processing system. All the computation for token authentication and FPGA reconfiguration is executed on the ARM CPU.

At the time of writing, no prior work shared an FPGA using tokens. There are many advantages to use a token-based framework. In fact, a token is a lightweight and standardized data structure that allows to efficiently communicate information. Such a data structure is easy to produce, communicate, read and act upon it. Hence, tokens provide low latency and lightweight solutions. For TokSek, JSON Web Token (i.e., JWT) is used as a token framework and library [118]. JWT is an open standard [119] for representing "claims" between two parties using tokens. The information within a JWT is signed, providing a means of verification and trust. The signing process can involve a secret with the HMAC algorithm or a public/private key pair (e.g., RSA).

JWTs are commonly employed in authorization scenarios where users access au-

thorized services, and resources after logging in. They are also utilized for secure information exchange between parties, ensuring the integrity of the claims made within the token. It's worth noting that while JWTs can be encrypted for added confidentiality, the focus in many cases is on signed tokens. These signed tokens verify the integrity of claims, and certifies the identity of the signing party.

The structure of a JWT comprises three parts such as:

$$token = \{header, payload, signature\}$$

The header specifies the token type (JWT) and the chosen signing algorithm, encoded in Base64Url format. This ensures that both token producing and consuming parties use the same methods and algorithms. The payload holds claims about the entity, categorized into registered, public, and private claims. The payload content can be extended according to specific needs. Both the header and payload are then Base64Url encoded. The final part is the signature, generated by signing the encoded Header and payload with a secret using the specified algorithm.

2.2 TokSek framework

TokSek is the implementation of the OAuth 2 adaptation presented in Section 2.1-1 with a software-based token endpoint deployed inside the FPGA. Later in Chapter IV, a hardware-based token endpoint for TokSek is proposed with additional elements to enable secure and multi-tenant FPGA acceleration.

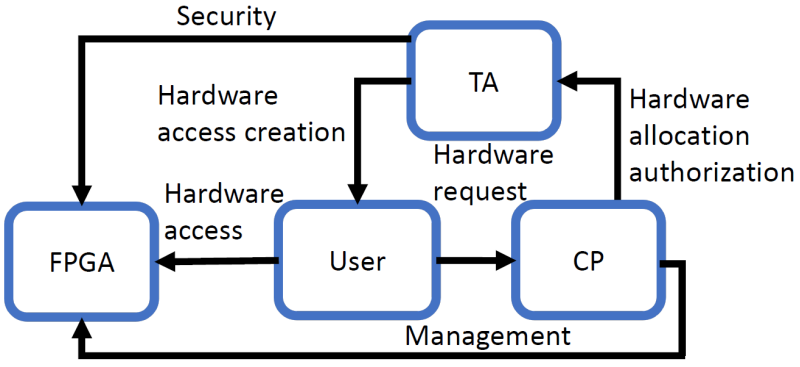


Figure 3-4 – High Level view of access delegation architecture including a trusted authority

A high level view of the TokSek architecture is illustrated in Figure 3-4 where the actions of each entity are shown. In this architecture, the user interacts with the CP by requesting hardware access. Then, the CP gives a hardware allocation

authorization to the TA by giving all the necessary information for the creation of an access token. The TA is responsible of giving the access token to appropriate user. The latter can then connect to the FPGA using the token. In this scheme, the CP is responsible of the management aspect of the FPGA by maintaining and deploying FPGAs. Additionally, the CP is the only entity that decides which FPGA to share and on which conditions. The enforcement method of these mechanisms are further described in the current section. Moreover, the TA is responsible of the security of the cloud based FPGA by deploying the HSM and setting up encryption keys.

2.2-1 Overview

In this proposal, a TA is involved with different mechanisms and requirements compared to the literature. In fact, the TA does not need access to the FPGA cloud after the deployment of accelerators. As described in Figure 3-5, the TA's involvement is in the deployment phase of the accelerator which is done offline from the rest of the framework. As shown in step 0, the TA is responsible for the certificate deployment of the FPGA device r_j. The certificate is signed by the TA and anyone connecting to r_j can verify this information to ensure the use of a legitimate r_j endorsed by the TA. To provide security to the hardware acceleration environment, the TA is responsible of the deployment of a Hardware Security Module (i.e., HSM) for each FPGA. The HSM can be open source since it does not contain any secret. This hardware security module can be compared to a TEE where cryptographic functions, random number generations and various security mechanisms are implemented. Chapter IV further describes the hardware architecture of the FPGA r_j and gives details about the functionalities present in the HSM. The only time the TA has to communicate with the FPGA is from step 0.1 to step 0.4. In fact, an FPGA shared secret (i.e., fss_j) is generated using a zero-trust online key generation protocol to establish a secure channel between an entity and a FPGA located in the cloud. This solution is called Linkguard and it is described in Chapter IV. This protocol is running inside the HSM located inside the FPGA fabric. The fss_j key is used by the TA to sign the FPGA connection information (e.g., access token described later in this section) of the user. The FPGA r_j must use fss_j to compute the signature of the access token to verify authenticity and integrity of the user connection information.

After key generation, fss_j is encrypted using the TA's private key (i.e., TAPrivKey) and a HMAC signature is computed for message integrity and authenticity. The TA can be sure that fss_j is not from a malicious entity. The TA must store one FPGA

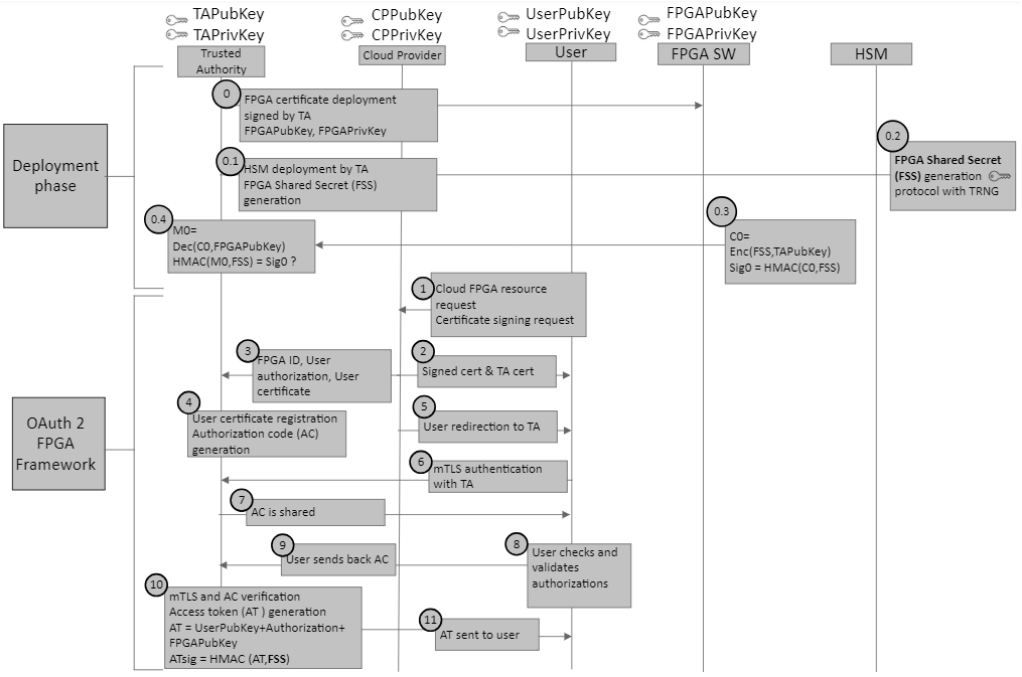


Figure 3-5 – The deployment and token generation phase of TokSek

identifier and one fss_j per r_j for further use. The TA does not need to store much information about one specific FPGA and repeatedly communicate with it. During the online phase of the framework, the TA only communicates with the cloud provider and the cloud user. This allows to involve a trusted third party with minimal security exposure and ensure low latency FPGA accesses. The processing time of user data is not impacted by this framework which is a major advantage of using FPGAs in cloud computing. Acceleration resources and software resources are independent and do not affect each other's performances. Additionally, the cloud user can benefit from enhanced isolation against the CP while having no exposure to the TA. The latter is not able to communicate with r_j and retrieve sensitive data after the deployment phase (e.g., fss_j key initialization).

In order to get r_j access, the user needs an authorization code from the TA. In step 1 of Figure 3-5, the user authenticates with the CP. User certificate and requested FPGA instance information are communicated to the CP. In step two and three, the user certificate is signed by the CP and the required information about the resource sharing is sent to the TA. These information are the user's certificate, the user's access scopes (e.g., $Perm(u_i) \in t_i$), an identifier of the allocated FPGA (e.g., $ID(r_j)$ and the reconfigurable region. These information are stored by the TA on step four and the user is redirected to the TA on step five for authentication purposes. Upon successful mutual authentication on step six between the TA and

the user, the authorization code is shared with the user in step seven.

At this stage, the user can request an access token with the TA. User authentication must be achieved and the authorization code must be presented to the TA as shown in step nine of Figure 3-5. Upon successful authentication, an access token is generated by the TA on step 10. This access token contains the necessary information to access r_j. To manage user authorization and allocate a specific FPGA, the CP shares information like FPGA identifier, instance identifier, and access duration. Then, the TA is able to identify r_j, authenticate it and use the appropriate shared secret fss_j to generate an access token with the CP's requirement. The token cannot be stolen and modified as shown in Section 1.2. It is encrypted and signed with HMAC (Hash-based Message Authentication Code) thanks to a shared secret fss_j between the TA and the user allocated r_j. As a result, the user u_i can connect to r_j with the access token t_i and use the FPGA according to rules set by the CP. The user does not go through this protocol if his access token is valid. The user does not need repeated authentication and resource allocation by doing these steps thanks to the single sign-on feature. The third-party FPGA cloud access can be permanent or temporary to withstand heavy network utilization spikes of a user. With a low latency access delegation framework, telco services could benefit from better flexibility, cost effectiveness and adaptability whereas cloud users could have a high Quality of Service (i.e., QoS) FPGA cloud access.

A user cannot communicate with r_j without an access token as described in Equation 1.2 in Section 1.2. A token cannot be shared, fabricated or stolen because the token is signed by the TA thanks to Equation 3-14. Additionally, the token is attached to a user certificate that is signed by the CP. Only the TA can create r_j accesses. Moreover, the token owner u_i is authenticated multiple times in order to get r_j access. The rules and policies set-up by the CP only apply to the token owner. As a consequence, a user who does not have an access token cannot access the resources allocated to another token owner. Other users will be rejected because they do not have a valid access token to communicate with r_j. All these claims are formalized in the model presented in Section 1.2.

2.2-2 User resource request and user certificate

To request a resource from the CP, u_i first sends a message to the CP's user-agent, as shown in step 1 of Figure 3-6. In this request message, u_i includes his identifiers, his certificate (or produces it online as stated below) and a redirection Unified Resource Identifier (i.e., URI). These information are sent to the TA in the next step. The

URI is used by the TA to send back redirected messages to u_i via the CP's user-agent. The URI represents a standardized addressing method crucial for uniquely identifying and locating resources on the internet. This identifier is foundational in web architecture, facilitating seamless resource navigation and accessibility.

There are two different scenarios for u_i certification. The user u_i can generate his certificate online or offline from the protocol. Online certificate generation proceeds as follows. Upon receiving the user resource request, the CP authenticates himself to the TA with his certificate, requests a certificate for the user and shares it with the user as shown in step 2 in Figure 3-6. The user certificate $signed\_cert(u_i)$ is created from the CP's website. The user u_i must interact with the web browser to create the certificate and add randomness to the generated keys. A similar mechanism is seen in Microsoft Azure's key-pair generation for SSH channel protection [13].

It is also possible to create $signed\_cert(u_i)$ offline from this protocol. The user has the responsibility of generating a key-pair for himself. By doing so, the user u_i makes the resource request to the CP with his certificate. Then, the certificate generation step is skipped and as a result the protocol is faster.

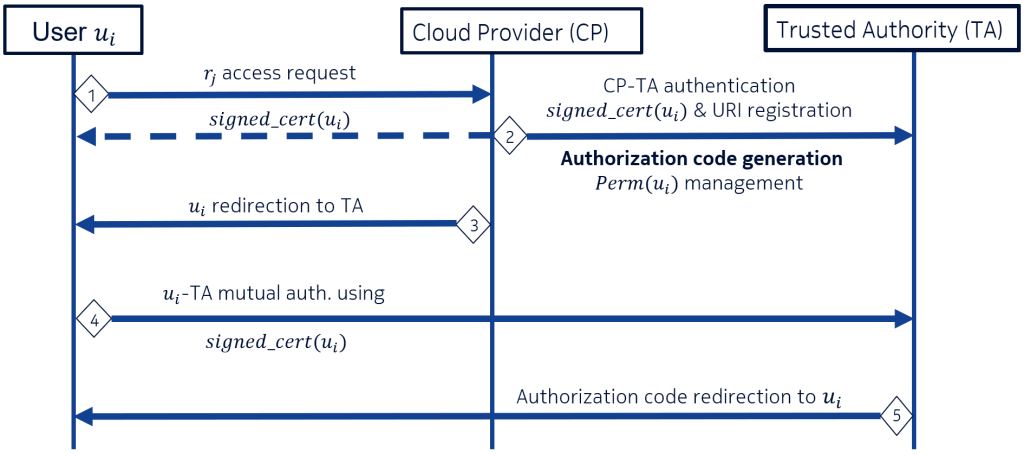


Figure 3-6 – CP introduces the user to the TA, generates and manages authorization code. User authenticates himself with the TA and obtains his authorization code.

This framework is based on a TA to create user-CP isolation by using an access delegation protocol. The TA has additional security sensitive tasks like r_j authentication, bitstream certification and verification. Next section shows the interactions between the entities present in our protocol to obtain the authorization code.

2.2-3 Authorization code grant

During the second step in Figure 3-6, after the CP's authentication, the user request is accepted or declined. If the request is accepted, the TA generates the authorization code. By using the previously provided URI, the TA redirects the CP's user-agent back to the user in order to authenticate him directly. By performing this action, the CP has authenticated and introduced u_i to the TA.

At this time, the TA knows $signed\_cert(u_i)$ and the authorization code associated with the user's identifiers. To obtain his authorization code, the user needs to authenticate with the TA for the first time. A certificate-based TLS authentication is performed [49]. If the user credentials are valid, the authentication is successful and the TA sends an HTTP redirection code to the CP's user-agent (HTTP code 302) alongside the user redirection URI. The user receives the authorization code from the CP's user-agent in the last step of Figure 3-6.

In this protocol, the authorization code cannot be used as an attack vector. In fact, the authorization code is associated with user credentials and the URI. The authorization code is not a secret code because the CP's user-agent shares it through HTTP redirection. The authorization code can be found in the user-agent history. In case of an authorization code redirection attack, which aims to get backdoor access to the user's resource, a simple redirection URI check from the TA is sufficient. The URI used when requesting the authorization code must match the URI used for the access token generation, as explained in Section 2.2-4. Hence, a malicious u_{eve} cannot gain access to resources attributed to u_i by intercepting the authorization code.

The role of this code is to ensure that the CP is authenticated and cannot be impersonated. By using the CP's user-agent to redirect the code, it can be confirmed that the CP which authenticated himself and gave authorizations for resource allocation to the TA is the same entity that communicates with u_i in the protocol.

2.2-4 Token generation and access management

After receiving the redirected authorization code, u_i needs to authenticate himself again with the TA using his certificate, his authorization code and redirection URI to request an access token from the TA. A certificate-based TLS authentication is performed one last time to confirm user identity. This second authentication is necessary to verify user identity.

As shown in Figure 3-7, the user needs to submit his authorization code to request

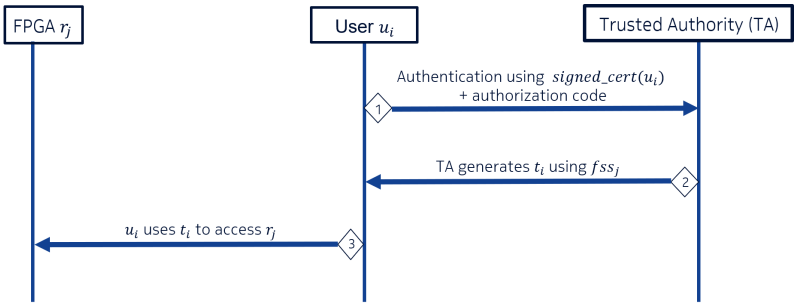


Figure 3-7 – The TA generates the access token for the user

the access token. The user needs to send the authorization code to the TA and proceed with a mutual TLS authentication. If the user credentials are registered and associated with the used authorization code, the TA will be able to construct a token t_i targeting r_j device allocated to the user by the CP. The TA generates the access token, signs it using a shared secret (i.e., fss_j) with the FPGA and sends it to the user on step 2. On the last step, u_i sends t_i to r_j to make use of the allocated resources.

Access tokens have scopes and duration of access (e.g., $Perm(u_i)$). They are managed by the CP and endorsed by the TA [55]. These options are requested by the user during the first step shown in Figure 3-6. Then, the CP accepts or declines the requested scopes and duration and notifies the TA (i.e., authorization server) in step 2. The user u_i can decline an issued token t_i if the obtained $Perm(u_i)$ does not match his requests. The access token's content may also be extended in order to contain information such as FPGA serial number, partially reconfigurable region identifier and so on. This feature gives flexibility for the implementation phase as additional mechanisms can be developed.

Using the described protocol, u_i is strongly authenticated through his certificate and his credentials with the CP and the TA. The authentication between the two last entities is not as critical as the user authentication because they are anchors in this protocol. Due to fss_j and the TLS session between the user and the FPGA, a secure and tokenized confidential remote access can be set up for the user. The CP is isolated from the user computation but can still manage access scopes and duration.

Efficiency and performance are constantly sought for accelerated computing. To follow this path, a user to user resource sharing mechanism that doesn't require the CP is proposed in the next section. A practical telco use case is described to emphasize the real-world application and relevance of this concept. This access delegation

solution can be used by a Telecom Operator (i.e., TO) to share his infrastructure with another TO. In a real use case, the latter would be a Mobile Virtual Network Operator (i.e., MVNO). The latter is a company that provides mobile services without owning the physical network infrastructure. Instead of building and maintaining its own cell towers and spectrum, an MVNO uses the network of an existing Mobile Network Operator (i.e., MNO). Essentially, the MVNO leases network access and then offers its services to customers under its own name. It's a clever way for companies to enter the mobile market without the massive investment required to build and manage a complete wireless network.

2.3 Access delegation between two TOs without CP implication

Telco architectures and systems are more and more complex. In 5G and beyond, cloud computing is a main component of the telco system [1]. Cloud computing solutions are deployed to offer efficient and reliable telco services. However, deploying and managing multiple large-scale cloud infrastructure is challenging and costly. A novel access delegation protocol is proposed to allow flexible and cost-effective telco services. This solution allows a TO to share its third party cloud resource access with another TO with minimal CP implication. There are at least five entities in this protocol : CP, TA, FPGA cloud, TO_1 and TO_2. This brings a new business and flexibility for TO_1 who has access to third party FPGA cloud resource. This use case is displayed in Figure 3-8. The TO has its own on-premise deployment and third-party cloud resource access shown with a blue arrow. The access delegation protocol allows the telco subcontractor (i.e., MVNO) to access and use the TO network as illustrated with red arrows.

In Open Radio Access Network (O-RAN), access delegation is especially important. Cloudification and interoperability are the key factors in 5G O-RAN. Some function of the 5G base stations (i.e., gNB) such as the Distributed Unit (DU) or Central Unit (CU) can be deployed in a cloud environment [120]. With the evolution of telecommunication standards, security requirements are stronger. The interoperability in 5G O-RAN introduced new security challenges. For example, the CU and DU from two different IP providers must work together without being a threat to the cloud architecture and to each other. A secure access to cloud resources is provided with the access delegation. As shown in Figure 3-9, a TA is involved in the access delegation scheme. Cloud resources owned by the CP are allocated to a telco provider thanks to the TA. Then, the TO can allow a MVNO to use its allocated

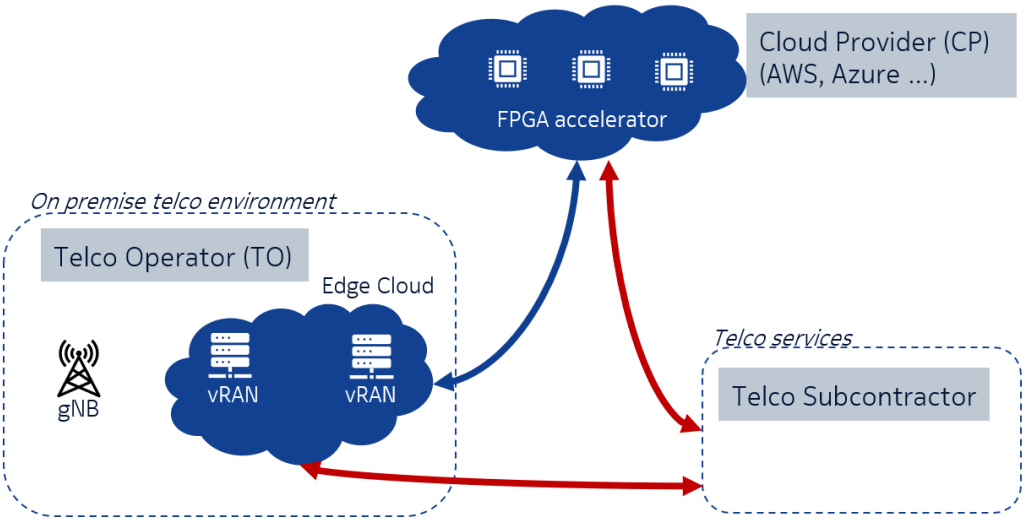


Figure 3-8 – Overview of the access delegation protocol between two TOs. The red link represents the child token access, the blue link represents the access obtained with the standard TokSek flow

third-party telco cloud. The MNVO has access to a telco cloud system without managing and deploying hardware or software. This architecture aims to create isolation between stakeholders to provide better security. The TA plays a key role in the access delegation process and its importance is discussed in Chapter II.

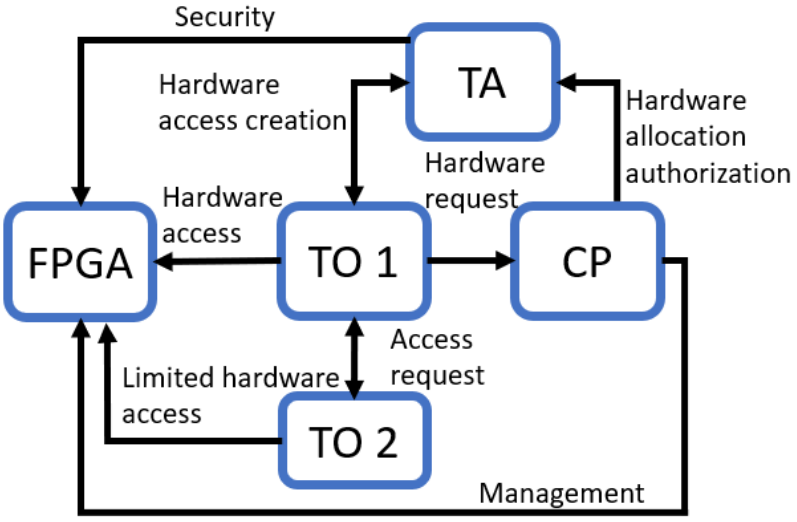


Figure 3-9 – High Level view of access delegation architecture including a trusted authority

Allowing an FPGA cloud user (e.g., TO) to share a subset of its allocated resources can bring great efficiency. Figure 3-10 details an access delegation scheme between two TOs. Only the first TO who gets access to the FPGA cloud can share

its resources with other TOs. Let TO_1 be the telco provider (i.e., MNO) who is willing to share the allocated resources and that has completed the protocol shown in Section 2.1-1. Let TO_2 be the MVNO who will get access to TO_1's resources.

Let the function *create_childtoken* defined such as:

$$\forall TO_1, TO_2 \in U, \forall t_{TO1}, t_{TO1,TO2} \in T, \forall r_j \in t_{TO1} / \\ create\_childtoken(t_{TO1}, TO_1, r_j) = t_{TO1,TO2} \mid \\ Perm(TO_2) \in t_{TO1,TO2}, Perm(TO_2) \leq Perm(TO_1) \wedge \exists signed\_cert(TO_2) \in t_{TO1,TO2} \quad (3\text{-}18)$$

Equation 3-18 describes the creation rules of the child token for TO_2. The latter only needs a signed certificate from the CP. The child token can only have permissions less or equal to the parent token.

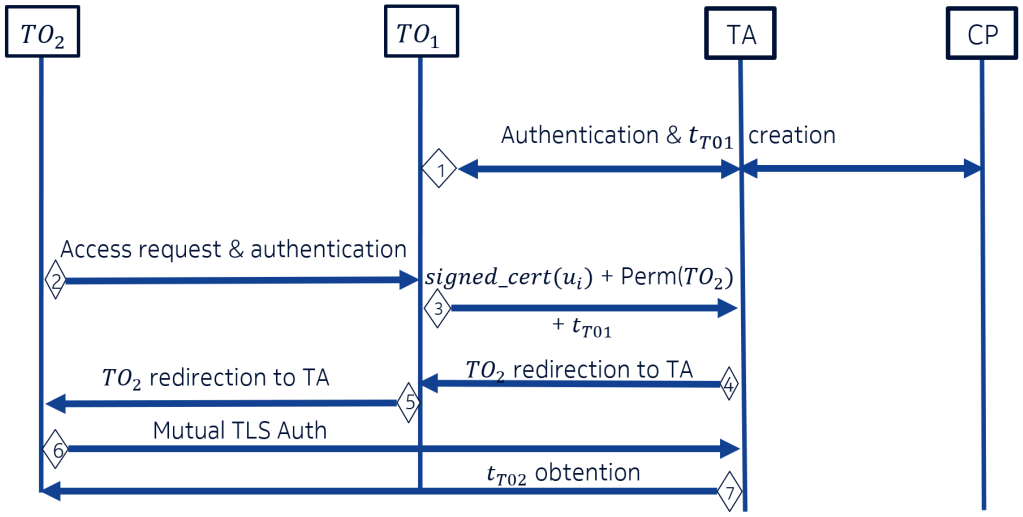


Figure 3-10 – Diagram for TO to TO FPGA access sharing

In Figure 3-10, step 0 is the initial access request and authentication of TO_2. On the first step of the protocol, TO_1's own access token (i.e., parent token), TO_2's certificate and the scope of the resource sharing are sent to the TA. TO_1 can only share resources that are allocated to it. As a consequence, TO_1 can either share the entirety or only a subset of its resources. After TO_1's authentication, the TA is able to determine what resources are allocated to TO_1 and which authorization it has. This prevents TOs from giving authorizations over resources not allocated to them. In this same step, TO_1 determines the authorization that TO_2 will get. TO_1 can restrict mechanisms like memory access or FPGA reconfiguration. TO_2 cannot get more authorizations than TO_1 under any circumstance. Then, the TA will create an access token for TO_2 using the rules set by TO_1 as described in Equation 3-

18. TO_2's access token (i.e., child token) contains information similar to the parent token. The child token is also a JWT token that contains the given authorization and the certificate of TO_2. The child token is constructed the same way as parents token and they possess properties described in Equations 3-8, 3-9, 3-10. On the second step, the TA requires TO_2 to authenticate itself in order to receive the child access token. On step 3, TO_1 redirects TO_2 with an authorization code towards the TA. On step 4, TO_2 and the TA are mutually authenticated and the authorization code is verified. The TA is able to authenticate TO_2 because TO_1 shared TO_2's certificate on step 1. On the last step, TO_2 receives the child token.

Thus the access delegation framework allows a TO to share third party cloud resources (e.g., FPGA) with other TOs. A cloud telco solution deployed on a third party CP can be shared.

While it is important to create a secure access to the FPGA resource, it is also important to have secure interactions with the FPGA. The following section introduces the mechanisms to offer a secure environment between the FPGA and the cloud user.

2.4 User-FPGA interactions

2.4-1 User and FPGA Secure Channel

After the token issuance, the user u_i contacts the FPGA r_j to obtain access to the allocated resources. A TLS session is set up for secure communication with perfect forward secrecy between the FPGA and the user [49]. The user and the FPGA create their shared secret with algorithms like DHE, ECDHE [49] and then use symmetric encryption algorithms like AES-256-GCM. Once the TLS connection is established, the user sends its token t_i to be authenticated. The FPGA parses the token and evaluates if the resources can be granted. Further communications between the user and the FPGA will be encrypted. User privacy is greatly enhanced and isolation from other entities is achieved.

2.4-2 Access Control with Tokens

When the entities are authenticated and the authorizations are granted, the user can access the FPGA with the access token using the $access(u_i, t_i, r_j)$ function defined in Section 1.2. The token needs to be authenticated by r_j with the function $auth\_token(t_i, fss_j)$. Actions are then taken by the FPGA to program and allocate

resources inside the device. Thanks to the resource server, the CP explicitly specifies the attributed resource information to the TA.

According to OAuth 2 protocol, token content can be extended according to user preference [55]. In order to take advantage of this feature in a cloud FPGA context, critical information needs to be selected. This information reinforces the access control of the user and ensures device/infrastructure security. It is up to the CP to decide the content of the token. In our solution, several pieces of information must be included. This information includes the FPGA serial number and a partial reconfiguration region (PRR) identifier in the case of multi-tenant FPGA usage. A user identifier (e.g., $signed\_cert(u_i)$) is useful for secure communication outside the FPGA. This information identifies the device, the user and the allocated PRR. Additionally, bitstream identification, and signatures are stored in the token. This aspect is further detailed in Section 2.4-3. The token must have validity timestamps for the FPGA to take action upon token expiration. This information is necessary to ensure that the user activity is located inside the cloud infrastructure and that the access delegation scope is respected. The TA should be able to create an access token with this information.

The CP's virtualization tools track which FPGA and PRR are in use. Cloud resource utilization is tracked by the CP and already allocated resources cannot be overwritten and reallocated by the user. Moreover, as the token has a limited validity, the FPGA should be able to end the connection with the user based on a timestamp without needing the CP to take action, the FPGA must be time aware. The access validity timestamp is available inside the token.

In the software implementation of TokSek, the only access control that is applied is for token ownership and validity. If $signed\_cert(u_i)$ is inside the token t_i, the user u_i is authorized to access and communicate with the FPGA r_j. All the permissions defined inside the token such as the memory size or the access to shared IPs are enforced by the HSM. These access control mechanisms are described in Chapter IV.

2.4-3 FPGA reconfiguration with bitstreams

Secure bitstream reconfiguration is another important topic for user data protection inside a third-party FPGA cloud. The user IP must remain confidential against the CP and other parties due to commercial value. Achieving IP confidentiality and protecting the FPGA cloud against remote attacks at the same time is a critical

challenge that needs to be addressed [121], [122], [101], [123], [124], [73]. On one hand, the CP must enforce IP verification on the user logic to protect the cloud infrastructure against harmful design patterns. These patterns and attacks are described in Chapter II Section 5.3. On the other hand, IP verification forces the user IP to be fully disclosed to the verifying entity.

In order to load a bitstream into an allocated resource, the user must fulfill several requirements. Bitstreams must be verified by the TA for malicious design patterns. The TA verifies if the bitstream created by u_i is only reconfiguring the user allocated $rr_{i,j}$. If the bitstream reconfigures illegal regions of the FPGA, the verification procedure fails. This verification step is sensitive for the user IP confidentiality. While, the bitstream is still exposed to the TA in plaintext, it is protected against the CP which is in agreement with the threat model described earlier. Then, the bitstream must be certified with the TA signature using the shared secret key (e.g., fss_j) associated to the user allocated resource r_j. The TokSek framework does not aim to provide a novel bitstream verification technique. Instead, this solution aims to provide a secure scheme by keeping existing verification methods used by cloud providers like AWS and Azure [3], [4]. In fact, the TA can execute bitstream verification because the TA has a shared secret fss_j with each FPGA, the signature and the bitstream verification can be trusted.

When the user tries to load a bitstream into his allocated PRR, the bitstream and the bitstream signature are shared as shown in step two of Figure 3-7. At the reception of the bitstream, the FPGA computes the signature and verifies its results with the signature sent alongside the token in step three. If the results are the same, the FPGA confirmed that the bitstream is certified as valid by the TA. The FPGA can proceed to reconfigure the user PRR in the last step.

3 Implementation and analysis of software-based TokSek

3.1 Implementation details

A Xilinx Zynq UltraScale+ MPSoC ZCU102 is targeted for this work. However, this board is not an optimal target for multi-tenant FPGA cloud acceleration and it is only used to demonstrate the proposed solution. Xilinx Alveo and Xilinx Versal boards are much more efficient and capable in that regard.

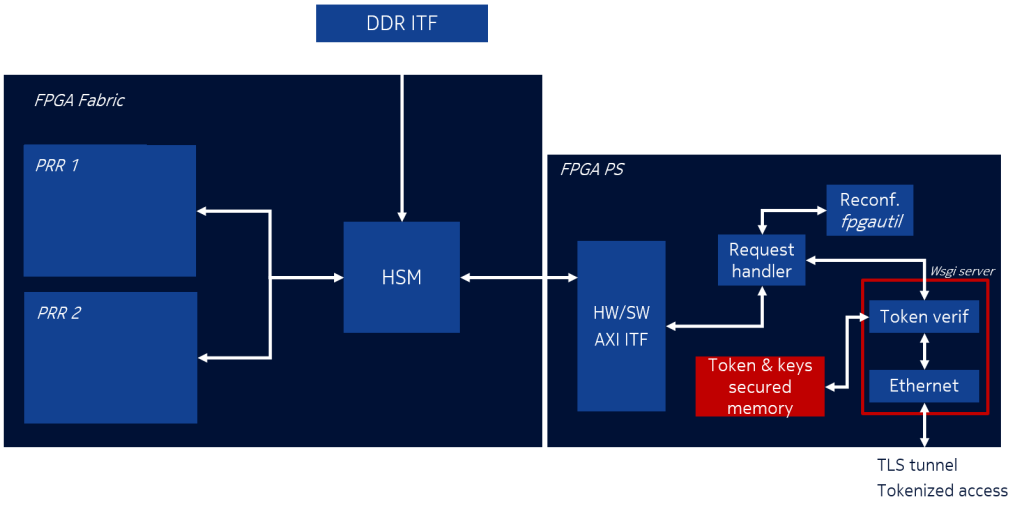


Figure 3-11 – High level view of the TokSek architecture

The ZCU102 has ARM cores for software execution and FPGA logic for hardware components. To take advantage from the software environment, an embedded Linux distribution with an OP-TEE (Open Portable - Trusted Execution Environment) overlay is deployed. OP-TEE is a trusted execution layer which implements ARM TrustZone technology [125]. This embedded environment brings a practical, privacy preserving and secure software stack to FPGA devices. Figure 3-11 shows the high level FPGA architecture. The Processing System (PS) is on the right hand side and the FPGA fabric on the left hand side.

In this implementation, the CP and the TA are implemented in VM_{CP} and VM_{TA}. The user is implemented in the device that hosts both VMs. The ZCU102 board is connected to the user device with an Ethernet cable.

On VM_{CP}, an Apache web server is deployed and few pages are served. This server is deployed by the resource owner (e.g. the cloud provider) as shown in Figure 3-3. A user can login and request an FPGA instance. When requesting an instance, the web server and the user go through a mutual TLS authentication. If the user certificate is signed by the CP, the protocol continues as described in Section 2.2.

On VM_{TA}, another Apache web server is deployed to serve an authorization server anchor as required in OAuth 2. The user that makes an FPGA instance request from VM_{CP} is redirected to VM_{TA} to receive authorization and access tokens. The web server of the TA also forces users to go through a mutual authentication scheme.

Inside the FPGA, a Flask web server is deployed to serve the OAuth 2 resource server. Flask is a micro web framework for Python that is particularly known for its lightweight nature [126]. Hence, it is a fitting choice for embedded devices with

limited resources. Flask's modular design allows developers to choose and incorporate only the components essential for their application, optimizing efficiency. Figure 3-12 shows a high level view of implementation described above.

The philosophy of this architecture is closer to the IBM FPGA architecture where each FPGA is independent of a host CPU [10]. Because we target an FPGA SoC, the CPU is leveraged to execute utility functions such as bitstream reconfiguration and to handle communications coming from the Ethernet port. Being independent from a host CPU can remove bottlenecks coming from the management of multiple FPGAs. An embedded CPU only manages its own computation whereas a host CPU can manage multiple CPU in parallel. Additionally, using FPGA logic instead of a software can diminish the attack vectors due to higher complexity to execute attacks. However, using embedded resources to deploy a lightweight server and compute encryption algorithms (e.g., due to TLS) can be penalizing because of performance constraints originating in the form factor of components (e.g., CPU, memory). This implementation will determine if the incurred computation overhead for running a flask server inside the FPGA is viable. Nonetheless, the proposed framework can also be implemented in a more standard host CPU+FPGA architecture that is more common in cloud deployments.

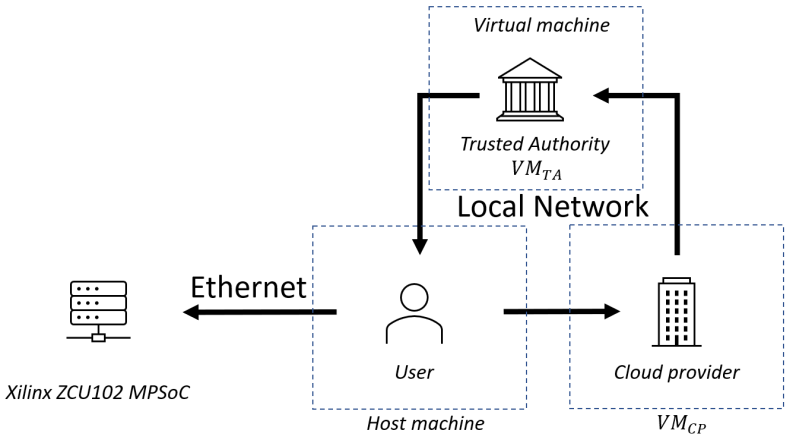


Figure 3-12 – High level view of the TokSek deployment

In the current software approach of TokSek, the modules inside the FPGA fabric are not considered. The primary focus in this current implementation is to access the FPGA with an access token and reconfigure an FPGA with a verified bitstream using *fpgautils* software drivers. The goal of this implementation is to determine the performance of this framework in terms of latency inside and outside the FPGA (e.g., CP and authorization server).

3.2 Theoretical Performance

To analyze this proposal, a theoretical model for different components can be built. All variables utilized in the equations below are described in Table 3-1. The required time to generate the authorization code as seen in Section 2.1-1 is described as follows:

$$\begin{aligned} t_{code} &= t_{CP}(internal) + t_{TA}(internal) \\ &+ t_{user-CP}(net. + auth.) + t_{user-TA}(net. + auth.) + t_{TA-CP}(net. + auth.) \end{aligned} \tag{3-19}$$

Internal tasks are computationally cheap. In this model, $t_{CP}(internal)$ is the time required by the CP's virtualization tools to determine available resources and allocate them to a user. Only simple read/write operations are executed by the TA to generate the authorization code. As a consequence, $t_{TA}(internal)$ is also computationally cheap. $t_{user}(auth.)$ and $t_{CP}(auth.)$ are the user and CP authentication latencies of the mutual TLS authentication. $t_{A-B}(net.)$ is the cumulated network and transport latencies between entities A and B. Assuming that communication latencies between entities are low (i.e., small $t_{A-B}(net.)$), the user can get the authorization code with low latency.

Once the authentication code is received the user can request an access token. The time required to get an access token as shown in Section 2.1-1 is described as follows:

$$t_{token} = t_{TA}(internal) + t_{user-TA}(net. + auth.) \tag{3-20}$$

t_{token} should be very small because there is only a certificate-based TLS authentication of the user and the TA. After the authentication procedure, the access token is generated by the TA. Few read and write operations are required for this operation. Every information required to create an access token is stored in the TA's database.

TO_1 now has an access token. The access token can be used by TO_1 to get FPGA access.

$$t_{access} = t_{user-FPGA}(net. + auth.) + t_{FPGA}(internal) << t_{code} + t_{token} \tag{3-21}$$

t_{access} is the time required to give TO_1 the allocated FPGA access. First, one TLS authentication between the user and the FPGA is needed. Then, the token verifica-

Variable	Description
t_{code}	Time spent to generate an authorization code
t_{token}	Time spent to generate an access token by using an authorization code
t_{access}	Time spent to connect to the FPGA using the access token
$t_X(internal)$	Time spent for internal computations by entity X
$t_{X-Y}(net.)$	Network latencies between entity X and Y
$t_{X-Y}(auth.)$	Time spent for mutual authentication between X and Y

Table 3-1 – Description of variables used in Equations 3-19 through 3-22.

tion function *auth_token* (see 2.4-2) must be executed. This latency is represented by $t_{FPGA}(internal)$. t_{access} is greatly smaller than the time required by our protocol to generate an authorization code and an access token. t_{access} requires less communications, whereas t_{code} and t_{token} include internal computing times and accumulated communication latencies between entities present in this protocol.

Additionally, TO_1 can create a child access token for TO_2 as described in Section 2.3. The time required by this procedure can be described as follows.

$$\begin{aligned} t_{u2u} = t_{TO_1-TO_2}(net. + auth.) + t_{TO_1-TA}(net. + auth.) \\ + t_{TO_2-TA}(net. + auth.) + t_{TA}(internal) \end{aligned} \tag{3-22}$$

t_{u2u} represents the time required to generate a child access token for TO_2 according to Figure 3-10. Same as Equation 3-19, entities mutually authenticate each other when starting a communication. The authentication procedures and the establishment of the TLS tunnel are a major part of this latency model. In fact, $t_{TA}(internal)$ is again small because it is the time to verify the received information and create a token as described in Equation 3-20. This time, it is TO_1 who authenticates and redirects TO_2 towards the TA. This duration is represented by $t_{TO_1-TO_2}(net.+auth.)$

The following section analyzes the security of the proposed framework to underline how the threats described in Section 1.1 are addressed.

3.3 Security Analysis

Thanks to the theoretical model proposed in Section 1.2 and the implementation, the following threats have been addressed.

User impersonation: Entities mutually authenticate each other before every communication. A user must have certificate signed by the CP to be authenticated. A MitM attack is heavily mitigated thanks to the mutual authentication. The CP

cannot impersonate the user too. A user needs an access token to use the allocated resource. The CP signs the user certificate but does not have the associated private key. The CP cannot impersonate the user and access the allocated resources.

Data integrity: To protect IP integrity, user bitstreams are verified and signed by the TA. The IP integrity is verified before IP and after TA verification. The verified bitstream is signed by the TA to prove the bitstream validity and integrity. Inside the FPGA, the bitstream integrity is verified before FPGA reconfiguration.

Confidentiality: To protect confidentiality, a TLS tunnel is used for every communication. User access token, data and IP are encrypted. They are decrypted inside the FPGA and sensitive information are stored inside TEE secure memory.

Logical isolation and access control: The access token represents the authorizations of a user with the allocated resource. On the user's first connection, authorizations are initialized by the FPGA according to the access token content. For each user request (e.g., Read/Write operations), access control rules are enforced. FPGA cloud users can only make requests that they are allowed to.

3.4 Experimental Performances

3.4-1 Access token request and resource access

Network latencies were not included in our testing environment to only measure the latency of our framework. Additionally, latency is dependent on user location and connection quality. For that reason, $t_{A-B}(net.)$ is not considered in the results section. With this implementation, we measure the average computing duration for one TLS handshake including mutual authentication. The average duration found is 34 ms. This means $3 \times 34 = 102$ ms for the computation of three mutual TLS authentication as noted in Equation 3-19. Moreover, $t_{CP}(internal) + t_{TA}(internal) = 183$ ms. In average, the access token is given to the user in $t_{code} + t_{token} = 102+183 = 285$ ms.

Then, the user connects to the FPGA with the access token. With the FPGA, the average duration for a TLS handshake with mutual authentication is measured to be 101 ms. The TLS handshake is nearly three times slower on the FPGA's PS compared to standard laptop CPU. Moreover, the internal FPGA tasks such as access token registration and user authorization initialization last 15 ms. As a result $t_{access} = 101 + 15 = 116$ ms without network latencies.

3.4-2 Third-party FPGA access delegation between users

After getting FPGA access, TO_1 can share its resource by contacting the TA. Required information are shared with the TA as described in Section 2.3. In Equation 3-22, $t_{TO_1-TA}(auth.)$ and $t_{TO_1-TO_2}(auth.)$ are measured to be 45 ms including mutual TLS authentication. Then, $t_{TO_2-TA}(auth.) + t_{TA}(internal)$ is measured as 52 ms. In total, the child access token is obtained in 142 ms without network latency.

3.4-3 Bitstream reconfiguration from the embedded OS

The user can send requests to the FPGA after being initialized inside the FPGA. FPGA logic reconfiguration is one possible request. The bitstream is sent by the user to the FPGA. According to [127], the maximum bandwidth for FPGA reconfiguration using the PCAP reconfiguration port is 500 MB/s. To reconfigure the FPGA, the *fpgautils* driver is used. The FPGA is reconfigured by a 26 MB bitstream in 204 ms. As a result, the observed bandwidth for FPGA reconfiguration is 127.4 MB/s. The same bandwidth is found in [60] by targeting a Zynq-7000 SoC. As a result, access token generation, user-FPGA authentication and bitstream reconfiguration is done in 503 ms with our implementation.

Moreover, the FPGA cloud environment proposed in work [44] is implemented and some metrics are measured. They grant an FPGA access to a user and reconfigure the FPGA with user IP (10 MB size) in 3.36s. FPGA reconfiguration time is linearly bound to bitstream size. In the present paper, a 2.6x bigger bitstream is used, resulting in a longer reconfiguration time. By using the linear property of the bitstream reconfiguration time, a 10 MB bitstream is reconfigured in 78 ms. With an internet speed of 6.3 MB/s (used in [44]), the network latency of the bitstream is approximately 1587 ms. With these constraints, FPGA access by a user and bitstream configuration is achieved in $503 + 1587 = 2090$ ms with our framework. Compared to [44], the proposed framework is 1.27 s (-37.8%) faster. Table 3-2 summarizes the latencies of this framework. Unfortunately, no other work propose an implementation of a secure FPGA access scheme to achieve secure and isolated acceleration.

4 Summary

This chapter provided a description of TokSek an its capabilities. Section 1 lays the foundation of the proposed framework by establishing a threat model and a formalization of the framework. The threat model conveys the goals and capabilities

Performance results of the proposed implementation	
Achievements	Computing time (ms)
Access token obtention	183 ms
FPGA access	116 ms
Child Token creation	142 ms
Bitstream configuration	204 ms

Table 3-2 – Performance Results

of the attacker that can be located inside or outside the FPGA. The formalization describes a theoretical model that specifies the rules and methods governing the TokSek framework. Section 2 introduces TokSek. First, the adaptation of OAuth 2 to FPGA clouds is discussed, then the token infrastructure and its strongpoints are explained. The protocol of TokSek is described in Section 2.2. Communications and computations of entities present in this framework are shown. Section 2.3 proposes an extension of the OAuth 2 mechanisms by allowing an authorized user to share his access token with another user independently from the resource owner (e.g., CP). A telco use case with O-RAN and 5G and beyond is given as a practical example. Results show that the token-based FPGA access solution overhead is small. In total, a user can obtain his access token and reconfigure the FPGA with a bitstream in 503 ms without considering network latencies that would exist in a real-world use case. One notable measure is the mutual TLS authentication between the FPGA and the cloud user. In fact, the FPGA is three times slower than a server deployed inside a virtual machine hosted by a standard laptop to execute a mutual TLS authentication.

Next chapter provides a hardware-based approach for TokSek. The software computation done inside the FPGA for token verification is offloaded to the FPGA logic to provide the first ever FPGA-based token framework for OAuth 2. The objective of this approach is to reach higher levels of QoS by reducing latency of token operations done by the CPU. The latter can be used for other tasks that require a software environment. Also, a hardware solution can achieve higher levels of security. In fact, sensitive computations like token authentication and resource allocation can be done in hardware logic independent of a CPU and the CP. Hence, the user data and sensitive operations can be isolated from the CP thanks to a hardware module with isolated memory enclaves storing crucial user information. As discussed in Chapter II, embedded CPUs and CPUs in general have cache related vulnerabilities that can

be exploited. Being less dependent of a CPU also means that FPGA accelerators without an embedded CPU can be secured with hardware logic. In this hardware approach, all the computations linked to token authentication, parsing and access control are done in the FPGA logic by the HSM. The latter is a crucial component of TokSek that initializes and enforces access control rules set by the access token. Linkguard is another significant hardware component of TokSek that is described in the next chapter. It is a patented solution to establish a zero-trust confidential channel with resources located in an untrusted cloud environment. Finally, the last hardware component of TokSek is an upgraded version of the ShEF shield module described in Chapter II. Linkguard is incorporated to the shield module to establish a confidential and isolated channel between the PRR region and the cloud user.

Chapter IV

Hardware approach for TokSek

This chapter proposes a more advanced hardware-based approach for TokSek. A solution involving a Hardware Security Module (i.e., HSM) is proposed to strengthen the security mechanisms in the FPGA fabric. The HSM serves multiple purposes. First, in TokSek, a hardware-based OAuth 2 is proposed where token operations like parsing and verifying are done by the HSM in the FPGA logic (i.e., resource server in OAuth 2). The framework can benefit from faster computing speeds, stronger isolation mechanisms towards the CP (e.g., memory enclaves) and CPU offloading thanks to hardware logic. Second, the HSM is responsible for enforcing access control and isolating tenants sharing the same acceleration platform. The permissions that are located inside the token must be read and applied inside the FPGA. The HSM must initialize the user FPGA context. For example, the HSM is responsible for memory allocation and a randomized memory allocator is proposed in Section 1.2-2. Third, the HSM serves as an access control tool for all PRR regions. When all the access control elements are initialized in the FPGA using the access, the HSM is ready to receive requests from PRR regions and apply access control rules. In fact, each PRR is connected to the HSM and all requests are checked to apply access control rules and other mechanisms like the address translator. Both of them are described in Section 1.3. Lastly, this chapter proposes a patented zero-trust confidential channel establishment solution for third-party cloud resources. This solution, called Linkguard, is proposed as an upgrade to the ShEF shielded enclave module [7] to provide a secure channel between the cloud user and its deployed IP. Linkguard is also included inside the HSM to establish a shared secret with the TA. This solution is described in Section 2. In Section 3, the shielded enclave is analyzed and vulnerabilities are exposed. It is a hardware module aiming to protect user accelerators inside the FPGA logic. We propose to upgrade the shielded enclave with Linkguard, and to use it under a modified scheme to address the vulnerabilities found. In Section 4, we expose the implementation results of the hardware modules in TokSek and we give a perspective on the results.

1 Hardware security module for token-based multi-tenant FPGA

The HSM is a hardware security module owned by the TA and deployed inside the FPGA logic of the CP. The HSM is responsible of essential functions to serve a secure multi-tenant FPGA inside the cloud. Figure 4-1 shows the architecture of the FPGA node with all of its TokSek components. There are three major components in the FPGA architecture of TokSek: Linkguard, the HSM and the upgraded shielded enclave. The illustrated architecture can accept multiple shielded enclaves (described in Section 3) to propose a multi-tenant environment. These modules are connected to the HSM, applying access control rules and ensuring isolation between entities.

In the current section, the components of the HSM are described. In this hardware implementation of TokSek, token operations are executed by the HSM instead of the PS. The hardware version of TokSek that is presented in this chapter is more advanced compared to the software implementation described in Chapter III. In fact, mechanisms like the resource manager, the policy store and the access control functions like *addr_translate* and *policy_check* are added in this implementation of TokSek to extend its capabilities.

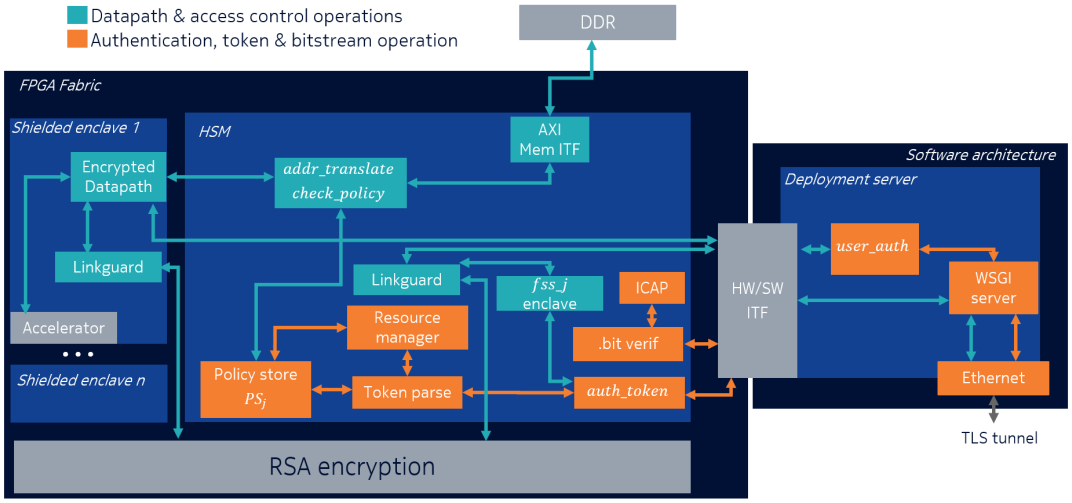


Figure 4-1 – An overview on the architecture of TokSek with all its components

The components inside the HSM are developed using High Level Synthesis (i.e., HLS). While it is not optimized regarding resource utilization, HLS allows to develop at faster speeds without handling the complexity of hardware development. For this implementation of TokSek, the PS is still used to deploy the flask-based python web server with mutual TLS authentication as described in Chapter III. Consequently,

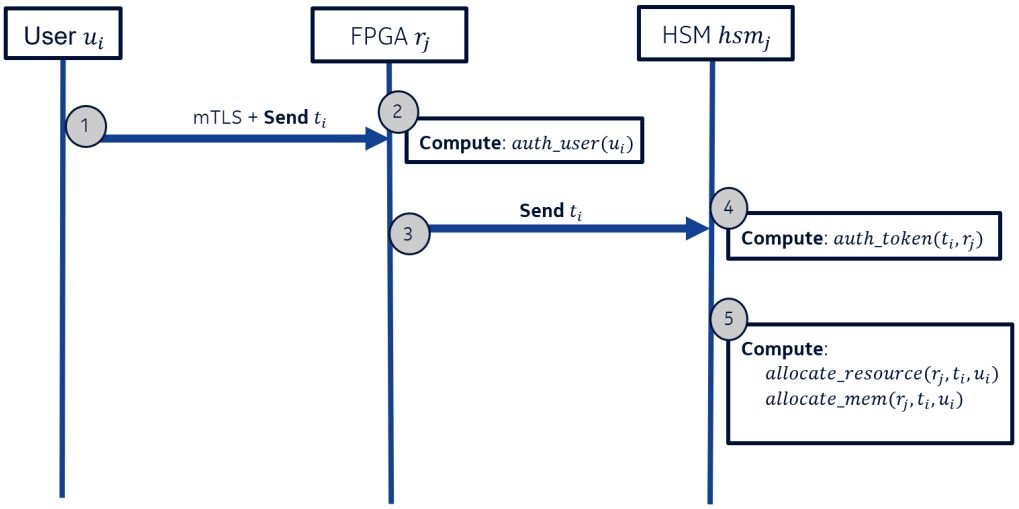


Figure 4-2 – Diagram of a user connecting to a target FPGA

only the *auth_user* function is executed in the PS with this implementation and all token operations are offloaded in hardware logic.

The HSM is responsible of all the token operations necessary to access an FPGA r_j and its resources. In this hardware implementation, the HSM executes functions such as: *auth_token*, *allocate_resources* and *allocate_mem*. The function *auth_token* described in Chapter III verifies the authenticity and integrity of the access token using software code. In the hardware implementation of TokSek, the function *auth_token* has been described in FPGA logic. In the current section, the functions *allocate_resources* and *allocate_mem* are introduced. Both of these functions are taking access tokens t_i as an input and they initialize the user policy store inside the FPGA.

Figure 4-2 shows the connection of u_i to r_j and displays the usage of core functions that are executed by the HSM.

1.1 Introduction of the HSM

The HSM deployed inside the FPGA logic is enforcing crucial security mechanisms to achieve a secure multi-tenant token-based acceleration environment. The capabilities of the HSM can be formalized with the properties below. This model uses the same naming convention as used in Chapter III.

1.1-1 Entities

User allocated reconfigurable regions $rr_{i,j}$ are identified by a shielded enclave $shield_{i,j,k}$ that is the k-th shield of user i inside resource r_j. The relationships between $shield_{i,j,k}$ and $rr_{i,j}$ are defined such as:

$$\begin{aligned} &\forall r_j \in R, \forall u_i \in U, \forall t_i \in T, \forall p \in \mathbb{N}, \forall k \in [1, \sum_{p \in \lvert er_{i,j} } \lvert rr \in er_{i,j,p} \rvert] \\ &\exists shield_{i,j,k} \in r_j \mid shield_{i,j,k} = \{rr_{p,j} \mid rr_{p,j} \in er_{i,j}\} \end{aligned} \tag{4-1}$$

The capabilities and objectives of $shield_{i,j,k}$ are described in Section 3. A user can deploy multiple shields to protect and isolate different modules inside its allocated reconfigurable regions. The user must have at least one shielded enclave per $rr_{i,j}$. If u_i has more than one reconfigurable region, the user then has k number of shielded enclaves as noted in Equation 4-1. The shield must be deployed in one of the reconfigurable regions owned by u_i. The shield is the only interface the user has with the HSM as shown in Figure 4-1.

To enforce access control rules on er_j, the HSM stores required user information in a memory enclave only accessible by itself. These user information include physical memory addresses, public or shared memory regions, access authorization for shared IPs and owned reconfigurable regions. Let ps_j be the policy store of hsm_j inside r_j that stores access policy information of its users. Each user policy $ps_{i,j}$ is stored inside the ps such as:

$$\forall i, j \in \mathbb{N}, \forall ps_j \in hsm_j, ps_j = ps_{0,j}, ..., ps_{i,j} \wedge \ ps_{i,j} = \{er_{i,j}, \{mbi_{i,0}, ..., mbi_{i,n}\}\} \tag{4-2}$$

Each user policy $ps_{i,j}$ is constituted with its allocated embedded resource $er_{i,j,k}$ and all of the associated memory block information set $\{mbi_{i,0}, ..., mbi_{i,n}\}$. The latter includes information like physical address, ownership, size, and memory confidentiality (private or shared memory).

1.1-2 Threat model

In this threat model, we consider that the attacker can be another tenant or the CP. Consequently, any module or interface of the FPGA can be compromised. The HSM is considered trusted because it is deployed by the TA. However, an attacker with CP privilege can compromise and reconfigure the legitimate HSM with a malicious one. Because the HSM is securing the FPGA logic, we also consider

threats inside the FPGA.

Confidentiality and access control: The attacker's objective is to steal valuable user data inside the FPGA trying to access victim memory region. The attacker tries to make concurrent accesses to shared IPs to obtain the data of the victim user. The attacker can try to impersonate the victim to access any resource allocated to him.

Integrity and denial-of-service: The attacker tries to modify the user data inside the device memory. By modifying the victim data, the attacker aims to create denial-of-service attacks where the victim cannot make use of his data. It is important to note that the CP do not want to harm its own infrastructure or degrade the quality of service of its cloud system.

1.1-3 *auth_token* and *token_parse* functions

In a software implementation, the token verification and parsing are straightforward because the JWT [119] token library has all the functions ready to use. Because such a library is not available for hardware logic, the token verification has been implemented in AMD Vitis HLS using C language. However, a JSON token parsing function already exists in AMD HLS library but it is compared against our implementation later in Section 4.

After the user u_i is authenticated by the FPGA software thanks to its signed certificate, the token t_i is sent to the HSM. In HLS, the access token is sent to the FPGA logic using the *hls:stream* type. Inside the hardware, this type is implemented as an AXI-stream protocol. The latter stands out from the AXI4 protocol, because it's designed for moving data in a serial way. AXI-stream uses signals like TVALID (valid data on the bus), TREADY (receiver ready for incoming data), TDATA (actual data transmission), and TLAST (end of a packet or frame). When we compare it to AXI4, which is good for burst transactions, AXI-stream is better at handling streams of data. AXI4 is more complex and has more control signals for synchronization purposes compared to AXI-stream. For example, some handshake signals like address write handshake (AWVALID, AWREADY) and write response (BVALID, BREADY) are used in AXI4 but not in AXI-stream. As a result, AXI-stream is less complex and offers better latencies compared to the AXI4 protocol.

The *auth_token* function has three major steps. First, after receiving the token t_i, it is parsed with the *token_parse* function to retrieve $< header, payload >$ data. The *token_parse* function is another feature that needs to be explicitly developed in AMD Vitis HLS. In fact, it is a JSON parser that allows to navigate around a JWT

token to search for key-value pairs. This function is used during *auth_token* and *allocate_resource* functions to find the appropriate information to process. In this section, each time we "search" or "retrieve" values inside the token, the *token_parse* function is used. Second, the token signature is computed by the HSM using a SHA256-based HMAC algorithm. During this process the signature is computed by using the fss_j key that is used by the TA to sign the token. Third, the computed signature is compared with the $< signature >$ data parsed inside t_i. If both signatures match, the token is verified. This verification is displayed in step four of Figure 4-2. After a successful token authentication, FPGA resource allocation happens on step five.

1.2 Resource allocation

1.2-1 Allocating the resources of u_i inside the HSM

On the first connection of u_i, there are no allocated resources and access policies set for users. After token authentication, the payload of t_i must be parsed and $Perm(u_i)$ must be retrieved to initialize user resources inside ps_j.

— **Property 1**

Let *allocate_resource* be the function employed by the HSM to allocate resources for tenants inside the FPGA. It is defined such as:

$$\begin{aligned} &\forall r_j \in R, \forall t_i \in T \\ &allocate\_ressource(t_i, r_j) = true \implies auth\_token(t_i, r_j) = true \wedge \\ &\forall k \in [1, |er_{i,j}|], er_{i,j,k} \in t_i \wedge vld\_until_k > current\_date \end{aligned} \tag{4-3}$$

The HSM allocates resources based on the user permissions $Perm(u_i)$ inside the token payload. Embedded resources $er_{i,j}$ are associated to reconfigurable regions $rr_{k,j}$. The user FPGA logic can access embedded resources (e.g., shared IP or memory) by making a query to the HSM. The latter can give access to $er_{i,j}$ after checking access control policies that are stored inside $ps_{i,j}$. Access control functions are further described in Section 1.3.

1.2-2 Memory allocation function

— **Property 2**

Let *allocate_mem* be an internal function of *allocate_resource* that randomly allocates memory according to the user token permissions. It is defined such as:

$$\forall u_i \in U, \forall er_{i,j} \in r_j, \forall t_i \in T,$$
$$allocate\_mem(u_i, t_i, r_j) = \bigcup_{k=1}^{|er_{i,j}|} mem\_region_{i,j,k} \mid allocate\_ressource(t_i, r_j) = true \tag{4-4}$$

And where $mem\_region_{i,j,k}$ is defined as:

$$\begin{aligned}&\forall u_i \in U, \forall er_{i,j} \in r_j, \forall t_i \in T, \forall m \in [1, |U|]\\ &mem\_region_{i,j,k} = \{(addr\_min_1, addr\_max_1), ..., (addr\_min_{frag}, addr\_max_{frag})\} \wedge\\ &size(mem\_region_{i,j,k}) = shrd\_mem\_size_k + mem\_size_k|\\ &shrd\_mem\_size_k, mem\_size_k \in t_i \wedge \nexists mem\_region_{m,j,k} \in ps_{m,j} \mid\\ &mem\_region_{m,j,k} \cap mem\_region_{i,j,k} \neq \varnothing\end{aligned} \tag{4-5}$$

Where,

- $mem\_region_{i,j,k}$ is a set of address couples that represents physical memory address blocks allocated to user u_i inside resource r_j. It cannot be attributed twice. In this model we make the distinction between physical and virtual addresses.
- Each couple in $mem\_region_{i,j,k}$ represents a memory block. The information of each block is represented by the memory block information set $mbi_{i,n}$ inside $ps_{i,j}$.
- $frag$ is the number of non-continuous memory region allocated to u_i on r_j. The memory fragmentation is represented further below and described in Equation 4-6.
- The total amount of memory allocated to u_i inside r_j must be equal to the total size of memory authorized inside t_i.

Figure 4-3 shows an example scenario with three tenants inside an FPGA that obtained memory allocations. The starting address of $mem\_region_{i,j,k}$ is randomly selected thanks to a TRNG. The first user in green has a single block of memory because he is the first user. Let's assume that the random starting address of the second user is smaller than the starting address of the first user. Consequently, the memory region in blue must be split in two in order to be fully allocated. And for the last user in red, an even smaller number is obtained by the TRNG. As a result the memory region colored in red must be fitted where any free space is available.

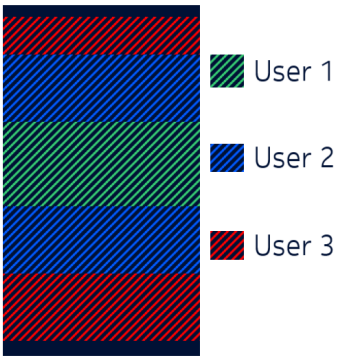


Figure 4-3 – Example of memory occupancy with three users

Then the memory is incrementally allocated to fulfill the user memory allocation. Determining a random starting address is useful to mitigate memory profiling attacks. With a random starting address, each user memory region is concealed and a user-specific address range cannot be targeted to retrieve data. Randomly selecting a starting address also mitigates Rowhammer attacks. It is a security vulnerability affecting Dynamic Random-Access Memory (i.e., DRAM) [128]. This attack exploits the vulnerability of DRAM with repeated and rapid access to a memory row. This type of interaction can alter adjacent memory cells due to electrical interactions between closely located cells. Consequently, "hammering" (e.g., repeatedly accessing) a particular row in the memory, the electrical effects can cause nearby rows to change their data. Rowhammer attacks have shown the ability to gain higher privileges on a system, break through virtual machine boundaries, and potentially compromise overall system security [129], [128], [130].

The *allocate_mem* memory function is executed inside the HSM by the resource manager module. The latter handles the resource allocation for users inside the FPGA. The allocated resources are noted inside the policy store ps_j inside the HSM. The proposed memory allocation function aims to fill the physical memory of the FPGA with dynamic allocation. To achieve this, the algorithm is inspired by the paging algorithm implemented for RAM in standard computers. Upon receiving a memory allocation request, the algorithm generates a random start address for the user and records the memory block information $mbi(i, n)$ in the policy store $ps_{i,j}$. The *allocate_mem* algorithm takes the random starting address and calculates the end address of the memory region to be allocated. The allocator finds and allocates free memory regions until the memory size inside t_i is fully reserved for u_i. For each memory region, a start address, an end address, whether it's a private or shared memory block with another user, and a fragmentation count number indicating which user memory region it corresponds to are stored inside the user policy store

$ps_{i,j}$. Initially the fragmentation count value (noted $frag$ in Equation 4-4) is zero, and with each fragmentation it increments by one. This value is crucial for locating a specific memory block owned by a user. In Figure 4-3, user 1 has a fragmentation count of zero whereas user 2 and user 3 have a fragmentation count of one.

The maximum number of $frag$ count of a new memory allocation can be calculated based on the number of existing memory blocks. In the worst-case scenario, a new memory block can be divided in $frag_{max}$ fragments to fit between the existing memory blocks according to the following equation:

$$frag_{max} = N + 2, N \geq 0 \tag{4-6}$$

Where N is the number of existing memory blocks. For example, in the worst-case scenario where three users have already allocated memory blocks and a fourth user needs a memory allocation after the token authentication, the new memory block can be fragmented into five pieces maximum.

There are two representation of $mbi_{i,n}$ inside ps_j: one for private memory regions and one for shared memory regions. The sizes of these memory regions are defined in $Perm(u_i)$. Both shared and private memory stacks are sorted by user, making them easy to access during address translation. After determining the origin of the request (e.g., $rr_{i,j}$, u_i) the corresponding physical address for user u_i can be easily found in the user sorted memory stack. A third memory stack of address exists to represent RAM usage and where each address block is sorted in ascending order. This third memory stack is useful for the algorithm to have a representation of what is already allocated in the RAM. It is used for allocating new memory regions. Because the stack is address sorted, finding unused addresses inside the stack is more convenient. The information of an allocated memory region is stored inside the policy store $ps_{i,j}$ to indicate the ownership of u_i. This is required to apply access control policies on memory accesses.

When adding a new memory region inside ps_j, the algorithm checks if it intersects with an existing region in the third memory stack. If so, depending on the type of intersection, the new block will either be moved or fragmented. Typically, if the starting address of the region to be allocated is entirely overlapping an existing block, the region to be placed is moved below the existing region. However, the new region is fragmented if it is partially overlapping with another occupied region. The non-overlapping region is allocated, and the overlapping region is moved further below to find a non-overlapping region. This procedure is illustrated in Figure 4-4.

After the execution of both resource allocation functions, the user resources are reserved and the acceleration environment is set. The *allocate_resource* function initializes the user authorization inside the policy store for shared IPs and stores other crucial information like the token timestamp. Then, the *allocate_mem* function allocates device memory to the user. The memory regions allocated to each user is stored inside the policy store ps_j.

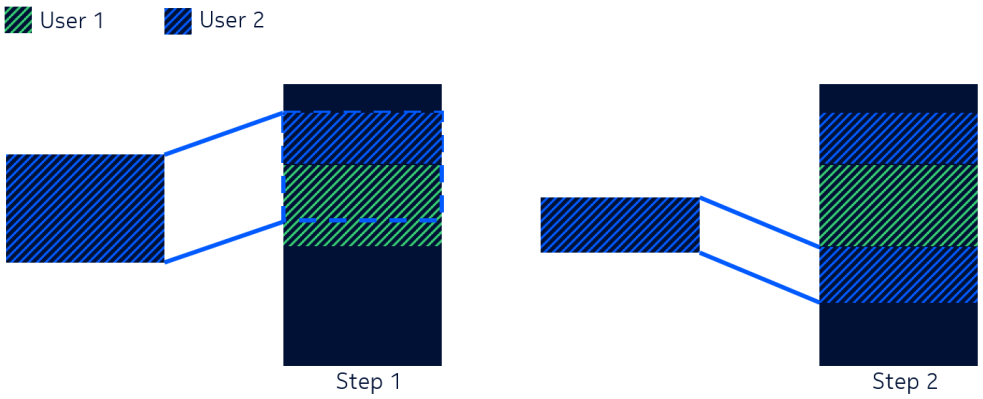


Figure 4-4 – Example of memory allocation procedure with fragmentation

If the user token t_i is no longer valid, or if the user terminates its acceleration environment, the resources allocated to u_i are freed and blanked such as:

$$\begin{aligned} &\forall u_i \in U, \forall er_{i,j,k} \in r_j, \forall t_i \in T, \forall k \in [1, |er_{i,j}|] \\ &deallocate\_user(u_i, t_i, r_j) = true \implies \forall vld\_until_k \in er_{i,j,k}| \\ &vld\_until_k < current\_date, mem\_region_{i,j,k} = \varnothing \wedge \forall rr_{i,j} \in er_{i,j,k}, rr_{i,j} = \varnothing \end{aligned} \tag{4-7}$$

The function *deallocate_user* removes user permission and blanks the previously used resources to the value 0 if the validity time for embedded resources $er_{i,j,k}$ is expired. The user reconfigurable region is reconfigured by a blanking bitstream to remove any remaining user logic. The user allocated memory is set to 0 to remove any residual data. To reflect this inside the policy store $ps_{i,j}$, any authorization linked to the expired eri, j, k is removed. The user u_i loses all access to expired memory region, reconfigurable region and shared IP. Doing these procedures allows to properly remove the authorizations from r_j without leaking data or keeping expired authorization.

The following section describes the method to access device resources securely while isolating each user acceleration context inside the FPGA.

1.3 Access control on allocated resources

In TokSek, the HSM plays a central role in the security of the multi-tenant acceleration environment. The HSM enforces access control policies to isolate the resources of tenants and secure their data. In fact, each tenant has its own FPGA logic protected by a shielded enclave. As shown in Figure 4-1, each user shield is connected to the HSM. The tenant logic must communicate with the HSM when it needs to access a device component like memory (e.g., private or public) or shared IP (e.g., third party accelerator or module). The HSM applies access control rules of appropriate users to resolve the user query and routes the user request to the corresponding component.

1.3-1 Policy verification function

User policies must be verified when accessing the FPGA resources and making requests.

— **Property 3**

Let the *check_policy* be the function that enforces access control rules that is defined as:

$$\begin{aligned}
&\forall u_i \in U, \forall rr_{i,j,k}, er_{i,j,k} \in r_j, \forall t_i \in T, i, j, k \in \mathbb{N} \\
&check_policy(u_i, shield, t_i, shrd_ip, r_j) = true \implies \exists shield_{i,j,k} \in er_{i,j,k}, \\
&\exists shrd_ip_k \in er_{i,j,k}, \exists vld_until_k \in er_{i,j,k} | \\
&shield_{i,j,k} = shield \wedge shrd_ip = shrd_ip_k \wedge vld_until_k > current_date
\end{aligned} \tag{4-8}$$

Equation 4-8 describes the verification procedure by the HSM for u_i possessing a token t_i to access a shared IP inside r_j. When u_i wants to send a command to a shared IP, the access token and its permissions must be verified against the user policy store. If t_i is valid and the permissions match, *check_policy* returns true.

Each $shield_{i,j,k}$ is associated to a token t_i that can give access to $shrd\_ip_k$ if the token validity period is respected. The HSM has a dedicated communication port for each $shrd\_ip_k$. By identifying the port on which the communication comes, the HSM can enforce the appropriate access control policy associated to the user owning the $shrd\_ip_k$ at that time.

1.3-2 Address translation function

The use of physical addresses would be too overwhelming for the user logic because of non-continuous address regions (e.g., fragmentation) created with the memory allocator proposed in Section 1.2-2. Additionally, users do not know what address ranges are allocated to them before using the access token for the first time as described in Equation 4-4. Consequently, the user cannot include physical address ranges inside the bitstream. Consequently, virtual memory addresses are leveraged to add a layer of abstraction for users. For example, the starting addresses of both users in Figure 4-4 are different. Thanks to the HSM and the *addr_translate* function, when each user request their first address (e.g.,0x01), the HSM finds the first address associated to the specific user.

— **Property 4**

Reconfigurable resources $rr_{k,j}$ can only access the device memory through the HSM. To add a layer of abstraction and enforce access control on memory accesses, address virtualization mechanism is leveraged. To make memory requests, each user utilizes virtual addresses. When receiving a request, the HSM uses the *addr_translate* function to translate the received virtual address into physical address. The HSM uses the translated physical address and forwards the user memory request to the memory interface. The $addr\_translate : (rr_{k,j}, virt\_addr_n) \rightarrow addr_n$ function that translates a virtual address $virt\_addr_n$ according to the reconfigurable region $rr_{k,j}$ can be described as follows:

$$
\begin{aligned}
&\forall u_i, u_m \in U, \forall r_j \in R, \forall t_i \in T, rr_{i,j} \in t_i, rr_{m,j} \in t_m, i, j, k, m \in \mathbb{N} \\
&addr\_translate(shield_{i,j,k}, virt\_addr) = addr_i \mid addr_i \in mem\_region\_(i, j, k), \\
&\sum virt\_addr_i = \sum addr_i \wedge \\
&\nexists virt\_addr \mid addr\_translate(shield_{m,j,k}, virt\_addr) = \\
&addr\_translate(shield_{i,j,k}, virt\_addr)
\end{aligned}
\tag{4-9}
$$

As opposed to the *check_policy* function, *addr_translate* is a memory request resolution function. It allows to make the memory request to the DDR interface by translating a virtual address to a physical address.

The following section formalizes attacker capabilities. It shows how TokSek and especially the HSM contribute to provide a secure multi-tenant acceleration environment inside an untrusted FPGA cloud. The following properties describe practical threats and how the proposed framework mitigates them.

1.4 The attacker perspective

Let's consider Eve and Bob being two tenants sharing the same FPGA r_j. Eve cannot access or modify any memory address owned by Bob.

— Property 5

$$\begin{aligned} &\forall bob, eve \in U, \forall r_j \in R, \forall t_{bob}, t_{eve} \in T, rr_{bob,j} \in t_{bob}, rr_{eve,j} \in t_{eve}, \\ &allocate\_mem(r_j, t_{eve}, eve) \cap allocate\_mem(r_j, t_{bob}, bob) = \varnothing \\ &\implies addr\_translate(shield_{eve,j,k}, virt\_addr_{eve}) \notin mem\_region_{bob,j,k} \end{aligned} \tag{4-10}$$

The HSM knows that $rr_{eve,j}$ is owned by Eve. Consequently, the *addr_translate* function interprets the memory access request according to the rules set by the HSM in $ps_{j,eve} \in ps_j$. Eve cannot modify the reconfigurable region $rr_{eve,j}$ allocated to her because PRR regions are statically defined. An FPGA user cannot decide to implement his logic in an area where the CP did not authorize. Consequently, Eve cannot impersonate the reconfigurable region $rr_{bob,j}$ because she cannot change her location inside the FPGA. Additionally, Eve cannot obtain a memory allocation that overlaps with Bob.

Moreover, Eve cannot specifically target the memory regions of Bob because the first address of Bob is randomly determined by using a TRNG. Eve knows how many tenants there are, but she cannot map the memory addresses to target a specific region due to the address translation function. It is worth noting that Eve also does not know her own memory regions because she uses virtual addresses to make memory requests. The ownership of physical addresses are only known by the HSM.

— Property 6

Any type of request made by Eve must go through the HSM to satisfy access control policies such as $check\_policy(u_{Eve}, shield_{eve,j,k}, t_{eve}, shrd\_ip_k, r_j) = true$. Eve cannot bypass the access control made by the HSM because $\forall rr_{k,j}$ there exists a hardwired physical connection with the HSM to reach other components inside the FPGA. There is no logical connection between any $rr_{k,j}$ and the device memory or shared IP.

As a consequence, Eve cannot bypass HSM access control to modify victim data inside the device memory.

— Property 7

Let's assume that CP_{eve} is the CP or has CP privileges. CP_{eve} cannot compromise the HSM or set up a malicious HSM due to Linkguard:

$$\begin{aligned} &\forall r_j \in R, \forall t_{bob} \in T, \exists fss_{CPeve} \in hsm_{CPeve} \mid \\ &fss_{TA} \neq fss_{CPeve} \Longrightarrow decrypt(t_{bob}, fss_{TA}) \neq decrypt(t_{bob}, fss_{CPeve}) \\ &\wedge decrypt(bitstream_{bob}, fss_{TA}) \neq decrypt(bitstream_{bob}, fss_{CPeve}) \end{aligned} \tag{4-11}$$

The deployment phase of the TokSek framework is described in Chapter III Section 2.2. In fact, the earliest step of the proposed framework allows the TA to establish a secret fss_{TA} key (e.g., fss_j through out this work). Because the user bitstream and user token are encrypted by the TA using fss_{TA} the malicious HSM hsm_{CPeve} cannot decrypt sensitive data like the user bitstream.

If we consider that fss_{TA} and the HSM are compromised, the only element that gets exposed is the user IP. Inside the FPGA, the user data confidentiality is still secure thanks to the upgraded shielded enclave. In fact, with the Linkguard upgrade on the shielded enclave, the user IP can process data and write encrypted data inside the memory. However, because the HSM is compromised, Eve knows the physical addresses of the user. But thanks to the shielded enclave, the data integrity can be verified. It is important to note that an attacker such as the CP is not interested in executing denial-of-service attacks on its own cloud infrastructure. Thus, Eve cannot make any use of the physical memory address allocated to Bob except for observing memory activity.

The following section describes Linkguard. It is a patented zero-trust key generation and confidential channel establishment module. Linkguard allows to set up an encrypted confidential channel between an FPGA logic deployed in an untrusted cloud and its owner. This module is an important element of TokSek.

2 Linkguard : Zero-trust confidential channel with cloud resources

2.1 Introduction

In FPGA accelerated cloud computing, the cloud providers cannot be trusted because they have physical access to the resources deployed inside the cloud infras-

tructure. CPs can deploy malicious tools to spy on cloud users. The cloud provider owns the hardware and the acceleration environment. This gives many privileges over the hardware that is shared with cloud users. The cloud provider has tools to compromise the cloud user's data and confidentiality. The software and the solutions deployed in the cloud infrastructure can be used to interfere with the cloud user's computation and operation. In FPGA, cloud user uses one or several IPs, defining user functions. The cloud user needs to establish authenticated and encrypted communication with its IP deployed inside an FPGA-accelerated cloud environment without exposure to third-party interfaces or software. The cloud user needs data and IP protection from the cloud provider to protect its interests. To ensure that, the user and the IP deployed inside the FPGA must have an authenticated and encrypted channel. The encryption keys and algorithms must be deployed independently from the cloud provider to ensure isolation and minimize the exposure. Additionally, access control rules must be established and enforced to communicate with the user IP. The cloud user must be the only entity that can access its IP. The cloud provider must not have the capacity to modify those rules and interfere with user acceleration session.

A cloud user cannot simply implement cryptographic functions and protect his data. It is not secure to embed any secret (e.g., encryption key) inside the accelerator IP because cloud providers have methods to retrieve plaintext information at various levels (e.g., device reconfiguration, device shell, etc.). Because the CP cannot be trusted, the cloud user must protect its interests without relying on mechanisms provided by the CP. The cloud provider has privileges over the deployed hardware. The FPGA shell and interfaces should be considered as a vulnerability for the user, and security measures must take that into account. The cloud end-user must be able to protect its data without using any third-party interface, software, or tool set up by the cloud provider inside the FPGA. This protocol must require zero-trust towards the CP to protect user data confidentiality and create a secure channel isolated from the CP.

Linkguard is a patented zero-trust confidential channel establishment solution for third-party cloud resources. It generates encryption keys using a True Random Number Generator (i.e., TRNG) inside an FPGA accelerator located in a cloud environment without involving the cloud provider. Using a hardware-based protocol independent from the CP, it creates isolation against other stakeholders and protects user data inside the FPGA as shown in Figure 4-5. In fact, the cloud provider owns the hardware, the infrastructure and is responsible of the deployment of the FPGA cloud computing environment. The cloud provider deploys a proprietary FPGA shell

to ensure communication and interfaces for the user IP. This can harm the cloud user's interest by interfering in the FPGA cloud acceleration environment. By generating encryption keys inside the hardware accelerator, optimal security and isolation is ensured. The FPGA fabric does not have an operating system or a runtime environment. Thus, software-based attack vectors are not valid in this context. The cloud provider cannot predict and access the random key generated in the FPGA because the cloud provider will not have interfaces with elements responsible of the secure channel establishment.

2.2 Modelization of Linkguard

2.2-1 Threat model

In this threat model, the attacker's objective is to break the confidentiality of the data processed inside the FPGA. We assume that the adversary has physical access to the FPGA and any memory that the user has access can be compromised. We assume that the CP and the TA may be malicious. Thus, solutions, protocols and software provided by these entities (except the HSM) may be inherently malicious. For example, any FPGA shell or interface provided to the user IP by the CP is considered untrusted. We assume that the shell is able to log and store communications and data belonging to the cloud user. Although the CP and the TA are both considered malicious, they do not cooperate to compromise the cloud user's data. However, the adversary is not interested in Denial-of-Service (i.e., DoS) attacks and attacks that would damage the CP infrastructure. As the FPGA is in the cloud, we assume that the cloud environment is untrusted. Consequently, any information that goes outside the protected IP must be encrypted to ensure data confidentiality.

2.3 Description of Linkguard

2.3-1 Key generation using a TRNG

To address issues presented in Section 2.1, an online key generation protocol using a TRNG is proposed. Linkguard is a hardware module deployed inside the FPGA logic to create a secure channel with other modules or entities. This present model uses the same nomenclature as the model used in Chapter III.

— Property 1

Each hsm_j inside r_j possesses a $trng_j$ that can produce a fully random key.

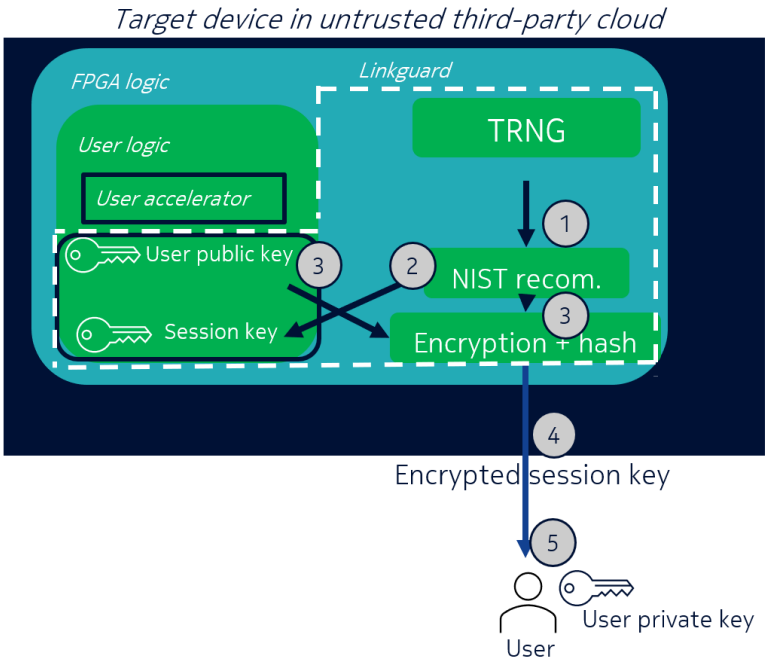


Figure 4-5 – Proposed online key generation protocol

This protocol is described in Figure 4-5. A TRNG is a logical circuit which exploits the physical randomness of the semiconductor to produce random bits that cannot be predicted. The physical randomness is originating from the silicium manufacturing process that cannot be controlled or replicated on purpose. The randomness embedded inside the semi-conductor creates small temperature and delay variations in electronic circuits. Knowing that, TRNGs leverage this irregularity to produce random bits. Because each FPGA semi-conductor is unique, each FPGA produce different random results for the same TRNG logic. Instead of embedding an encryption key inside a device, a TRNG can be implemented inside a cloud-based FPGA. By doing that, no keys are exposed to the CP inside the user FPGA bitstream.

The secure key generation protocol is described in Figure 4-5. The protocol has four blocks. The first block is the TRNG logic. In this work, a Transition Effect Ring Oscillator-TRNG (i.e., TERO-TRNG) is implemented [131] and its architecture is shown in Figure 4-6. This TRNG design is composed of a TERO cell, a sampling clock and two D flip-flops. The TERO cell is a loop made of an even number of inverters gates and any number of buffers. Inside this loop, there are two propagating signals and one is faster than the other due to the delay difference of inverters and buffers. The randomness created by the TERO cell feeds the first D flip-flop which is a modulo two counter. Then the counter is sampled by the ring oscillator with the second flip-flop where the output is the random bit.

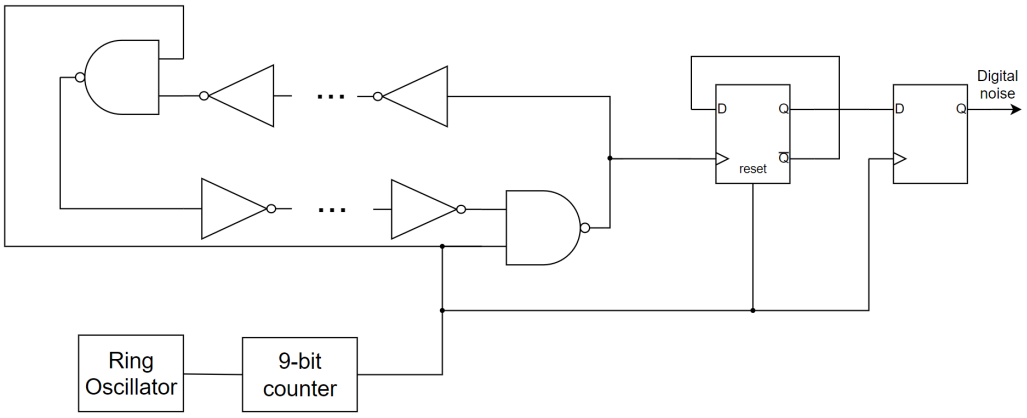


Figure 4-6 – Architecture of a TERO-TRNG

In Linkguard, producing a random key is the first step as shown in Figure 4-5. Generally the Linkguard procedure must be the first task after IP reconfiguration so that a user can establish a secure communication link with his IP.

2.3-2 NIST recommendations for TRNG key generation

In step 2 of Figure 4-5, the produced TRNG key is transformed according to the NIST recommendations [132]. In fact, a TRNG key should not be used as such. Instead, the random output of the TRNG must feed a Deterministic Random Bit Generator (i.e., DRBG). It's a deterministic algorithm designed to produce a sequence of random bits based on an initial seed value. The necessity to use DRBGs originates from the critical role randomness plays in various cryptographic applications. Unpredictable randomness is essential for generating secure keys and initialization vectors. Unlike TRNGs, which extract randomness from physical processes, DRBGs are deterministic, meaning that if the same initial seed is used, it will produce an identical sequence of random-looking bits. This determinism is advantageous in cryptographic contexts, as it allows for reproducibility and consistency in generating cryptographic elements. DRBGs typically consist of an entropy source to gather initial randomness (e.g., TRNG), a seed generation algorithm, and a pseudo-random generator that transforms the seed into a longer sequence of bits. The periodic reseeding of DRBGs helps maintain their security over time, ensuring the unpredictability of generated output.

In our solution, 440 random bits of TRNG output are required to produce a 256 bits symmetrical key according to the NIST recommendation [132], [133]. The key coming from the TRNG feeds a DRBG to produce the final key. In this FPGA implementation of the NIST DRBG recommendation, we used the hash-based DRBG

to create the session key. In the NIST documentation, the DRBG algorithm has two steps to generate the final encryption key. First the *DRBG_instantiate* function uses the 440 bits TRNG key to create intermediate variables like initialization vectors using a hash function (e.g., SHA-256). Next, those variables are used in the *DRBG_generate* function to produce the session key. After the execution of both functions, the generated key is stored inside the user IP in step three of Figure 4-5. The diagram shown in Figure 4-7 shows the communications and computations between the Linkguard components. Next, the encryption needs to be communicated back to the user to set up the encrypted channel.

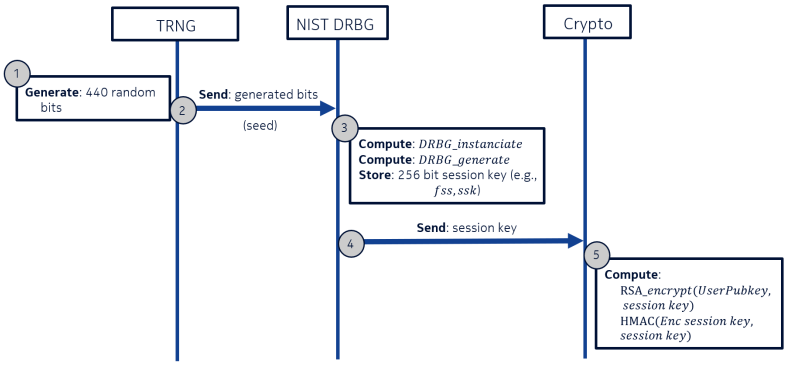


Figure 4-7 – Proposed online key generation protocol

2.3-3 Encryption and communication

The generated key cannot leave the Linkguard module without encryption. To communicate the key to its owner, it must go through the FPGA interfaces, and use the FPGA ports which are both considered unsecured. To protect the key against these components, we embed the public key of the user inside the Linkguard module. There is no harm in embedding a public key inside the shield because the private key is remaining secure and is not communicated. To protect the key integrity and confidentiality, a HMAC function is used alongside RSA encryption. The session key is encrypted with RSA using the user public key. Then, the encrypted session key is signed with itself. The session key is sent to the user and it is decrypted with the associated user private key. The IP protected by Linkguard and the user are now sharing a secret symmetrical encryption key. The data sent by the user is encrypted with the shared key and it can only be decrypted by IPs. At this point, the cloud provider can only log encrypted data when the user is sending and receiving data from his IP.

2.3-4 Key reception and protection

Upon receiving the message on the last step, the user can use their private key to decrypt the encrypted session key sent by the FPGA. Then, the cloud user must verify the signature of the encryption key against the signature received in the message. To verify the session key integrity, the cloud user must implement the same HMAC algorithm that is used inside the FPGA. The signature and the session key decryption must be executed without exposing the data and the results to the cloud provider. If the verification is executed in a software environment inside the cloud, enclave technologies like ARM TrustZone or Intel SGX must be used to ensure data protection against the cloud provider. However, software enclaves still have vulnerabilities. Hardware computing is still considered more secure thanks to the lack of a runtime environment which reduces the attack vectors. However inside a hardware accelerator (e.g., FPGA), the session key must be secured by keeping the sensitive information inside the FPGA fabric without being exposed to external interfaces.

Nonetheless, the security of the session key verification depends on the two endpoints that need to setup a confidential communication channel. In fact, the security requirements for the verification and storage of the session key are not the same if the computation happens inside or outside the cloud. Because the session key must remain secret, storing it inside a cloud solution or inside the FPGA memory in plaintext is not secure. However, storing the session key inside the FPGA logic (e.g., HSM or user IP) is more secure than DDR or cache memory due to the logical isolation created by the hardware environment. For these reasons, Linkguard is effective to secure the communications between FPGA logic and external stakeholders.

In TokSek, Linkguard is used by the TA and by the user for different purposes. In this context, the TA implements Linkguard inside the HSM to create a shared secret (e.g., fss_j) with the device r_j. This allows to create access tokens t_i for users u_i that are signed with fss_j. As a result, the HSM is able to authenticate tokens and establish a secure FPGA access. Additionally, the user can use Linkguard to secure the data produced by his FPGA logic. As a consequence, the FPGA user can protect his data and can securely store and communicate it outside the FPGA by using a session key produced inside the FPGA logic. The use cases for the user IP and the HSM are both described in Section 3 and 1 respectively. The next section provides a practical use case for Linkguard and how it allows to secure telco applications.

2.4 Practical application example for Linkguard

Cloud providers will be central component of Open RAN (O-RAN) architectures. 5G and beyond continues to evolve, enabling new network solution. O-RAN is one of the biggest 5G evolution, expanding ecosystem of mobile phones and RAN players in telco network market. Opening interfaces between telco network elements allows multi-vendor interoperability. As a result, multi-vendor telco network solutions, in different network domains (RAN, Core Network (CN)) will be built by telco operators. To reach latency and performance constraints for telco O-RAN networks, hardware accelerators (e.g., GPU, FPGA) are embedded in the different O-RAN network elements. Hardware accelerators allow to reduce number of required CPU cores, especially for L1 processing and compute-intensive workloads (AI/ML, cryptography applications, etc.). Hardware accelerators can use different technologies, including FPGA, ASIC and GPU. Originally embedded to accelerate AI/ML applications, FPGAs are relevant candidate execution platforms for O-RAN. Compared to on-premises network solution, in which an operator has its private network, O-RAN openness enables innovation acceleration. O-RAN can use shared network, including commercial cloud environment. O-RAN brings new security challenges, which is critical in telco networks. O-RAN Security Work Group (or WG11) has been created to ensure O-RAN system security. O-RAN WG11 security requirements are presented in [134]. O-RAN security threat modeling and remediation analysis are highlighted in [135]. Risk assessment from Table 7-2 in [135] presents the O-RAN risks, impact, severity, and likelihood level. Several threats are linked to hardware accelerator usage (Threats Id: T-AAL-01, T-AAL-02, T-VM-C-05, T-HW-01, T-HW-02).

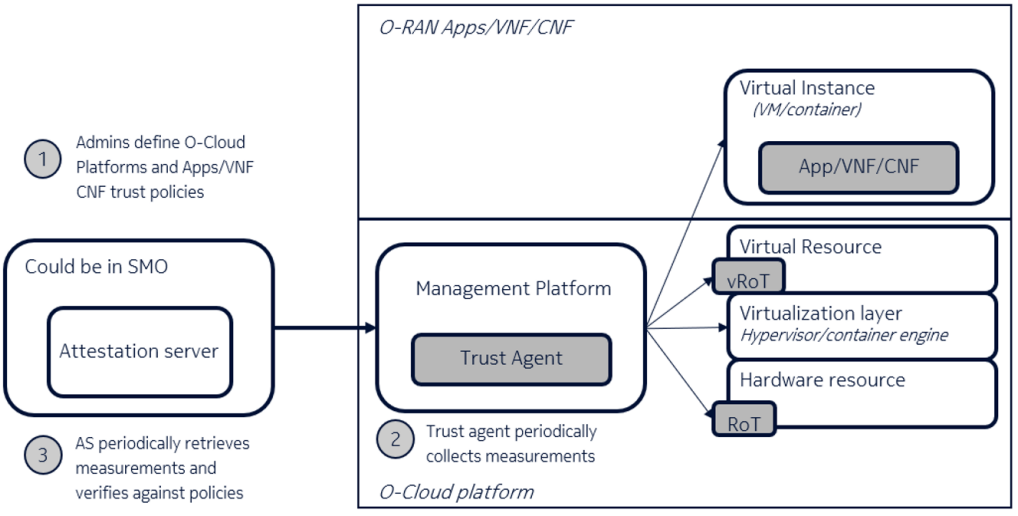


Figure 4-8 – O-RAN hardware security architecture overview

FPGA usage in O-RAN context is similar to FPGA-cloud computing solutions. FPGA is used in cloud environment network, sharing their resources between end-users (operators) and evolving as a function of time (through reconfiguration). One critical issue linked to FPGA usage in cloud context is the lack of integrity solutions to ensure security. If different platform integrity technologies solutions exist for CPU-based platform (e.g., Arm TrustZone, Intel SGX, RISC-V PMP), no similar solution is proposed to protect FPGA platforms.

Linkguard addresses the data protection and confidentiality in O-RAN FPGA accelerated cloud context. With FPGA accelerators inside cloud environment, the end user expects security and confidentiality. To save capex and opex, O-RAN based network will generally access accelerator capabilities through Cloud Provider (CP) (Amazon, Microsoft, IBM. . .). As already discussed, in this scheme, security limitation currently exists. CP can access exchanged data between end-user and accelerator. No encryption is realized between the hardware accelerator and other entities inside the cloud infrastructure. For example, in O-RAN, hardware accelerators are deployed using Kubernetes or Openstack. The interfaces to hardware accelerators are not encrypted. Applications, and the hardware accelerator manager are sending and receiving plaintext data from the accelerators. This is not specific to O-RAN. In public clouds such as Amazon AWS EC2 F1 and Microsoft Azure, the data coming in and going out of the hardware accelerators are exposed. Cloud providers deploy FPGA by developing their custom software and hardware layers. These layers can include monitoring, management, and logging mechanisms. In Amazon AWS EC2 F1 acceleration environment, a logic FPGA shell is implemented by Amazon. The shell implements management and application functions (e.g., PCI communication and OpenCL API calls are handled by the shell). The shell also has master and slave interfaces with the user acceleration logic. Because of a lack of transparency, the whereabouts of the cloud provider are unknown. The cloud end-user does not have any mechanism to verify the confidentiality and the security of his FPGA acceleration environment. In this context, Linkguard does not need any mechanism to verify the security of the acceleration environment. Linkguard does not need a trusted environment to be deployed and it can be considered as a zero-trust solution.

An overview of the hardware security management in O-RAN and the Open Cloud (i.e., O-Cloud) are detailed in Figure 4-8. O-Cloud is the open cloud environment where different modules, applications, Virtual Network Functions (i.e., VNF) and Containerized Network Function (i.e., CNF) are deployed [136]. In O-RAN, trust policies for applications and VNFs are defined in the Attestation Server (i.e., AS) by system admins. Attestations are produced for each module by the AS

using the policies described earlier. Attestations are deployed for various modules inside the O-Cloud. These attestations describe each module's authorization and capability. A management platform is present in the O-Cloud and a trust agent is deployed. Measurements of the Root-of-trust (i.e., RoT) and virtual RoT (i.e., vRoT) are periodically collected. A RoT is a form of embedded trust inside the hardware like the firmware version, hash value of software code or engraved device keys. In O-RAN security requirements, the chain of trust must start from the hardware firmware and chain of attestation must be formed with virtualization layer and the App/VNF/CNF layers. If one layer of the attestation chain is corrupted, then the whole system is considered untrusted and corrupted. But once a secure channel is established using Linkguard, the communications remain secure even if the chain of trust is broken.

In current O-RAN, if the chain of trust measurements is not valid when verified in the attestation server, the RoT must be reestablished to secure the system. This protects the O-Cloud against tampering and rogue behavior. One drawback in this solution is the lack of data confidentiality between the hardware accelerator and upper layers. The communication channels with the hardware accelerator are not secured. Consequently, the virtualization layer and the virtual machine can retrieve information from the hardware accelerator and the virtual resource [137]. They can create logs and spy on the ongoing App/VNF/CNF. This can be done without tampering the chain of trust. Hence, the measurements are not modified, and no flags are raised by the AS. Data confidentiality of communications between the hardware accelerator and other modules must be ensured. This security limitation is not permissible in O-RAN network context. Linkguard is a zero-trust end-user solution for FPGA-based cloud architecture in O-RAN context. It enables isolation between end-user, CP, and accelerator.

Linkguard solves threats on hardware accelerators about data confidentiality and protection in O-RAN. This solution mitigates threats IDs T-HW-01, T-HW-02, T-HW-AAL, T-O-RAN-01, T-O-RAN-02 in [135].

This section provided a description of Linkguard and a comprehensive use case for a practical application like O-RAN. Linkguard is a zero-trust key generation protocol that allows to establish secure channels between two entities. This solution is particularly interesting when used to establish a secure communication link with a resource located in an untrusted cloud environment. Because, third-party cloud providers can be intrusive on user privacy, Linkguard is a good example of secure communication and data protection mechanism independently from the cloud

provider. The following section introduces a shielded enclave solution that aims to protect the user accelerator inside the FPGA against external modules. The shielded enclave has been analyzed and security vulnerabilities are exposed. An upgrade with Linkguard is proposed to address the security challenges of the shielded enclave. The shield enhanced with Linkguard is an important component of TokSek that provides IP security and data protection against malicious entities.

3 Shielded enclave for FPGA logic for secure acceleration

In Chapter II, a shielded enclave for FPGA cloud accelerators, addressing data protection in public cloud environments is presented [7]. The architecture involves the CP, Data Owner (i.e., DO), and IP vendor. The IP vendor creates an asymmetric shield encryption key, shared with the DO. A symmetric data encryption key is generated by the DO for secure communication with the FPGA. The shield IP encrypts outbound and decrypts inbound data using the AES block. Additional security measures include digest algorithms, a security kernel, and a security processor block. The final FPGA binary is encrypted with a bitstream encryption key. Multiple shielded enclaves support spatial FPGA multi-tenancy.

In this section, a vulnerability analysis and an upgrade to the shielded enclave are proposed. By adding Linkguard to this solution, the security drawbacks of the shielded enclave are mitigated.

3.1 Shielded enclave mechanisms

3.1-1 Outside the FPGA

The scheme proposed in [15] aims to protect the confidentiality of the FPGA IP sold to the data owner. As shown in step three of Figure 4-9, the IP vendor negotiates a session key with the FPGA security kernel. This key is useful to establish a secure channel between the FPGA and the IP vendor. This channel is used to send the bitstream key (i.e., BtstrmKey) to the security kernel. The latter needs it to reconfigure the FPGA with the encrypted bitstream.

However in this work, the user data privacy is the primary concern. To address this issue, each shielded enclave has an asymmetric key pair noted as $ShieldEncKey_{pub}$ and $ShieldEncKey_{priv}$ in Figure 4-9. These keys are generated by the IP vendor and

allow to securely communicate with a shield inside an FPGA. To protect his data, the data owner generates a data encryption key noted $DataEncKey$. The latter must be encrypted by $ShieldEncKey_{pub}$ and sent to the shield to be used inside the FPGA.

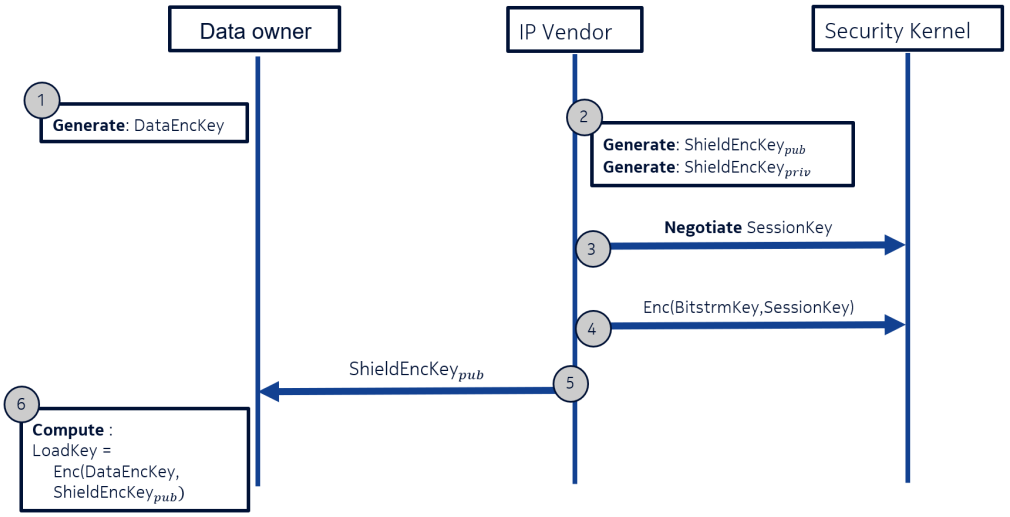


Figure 4-9 – Generation of the load key

3.1-2 Inside The FPGA

In [15] the shield has two purposes: protect the IP of the IP vendor and protect the data of the data owner. In this section, we are only interested in securing the data of the cloud user. As shown in Figure 4-10, the shield is placed between the accelerator and the interfaces of the FPGA that are not trusted (e.g., FPGA shell). In FPGA cloud context, an FPGA shell is a non-reconfigurable hardware logic deployed by the CP to provide management and monitoring functions inside the FPGA. For example, the AWS FPGA shell described in [3], shows a management and application physical function. A cloud user can utilize shell interfaces to access memory, use device clocks, resets, etc.

One achievement of the shielded enclave is the I/O isolation of the user accelerator against the outside. The user IP is protected by a "shield" that do not allow malicious request to go through. This is achieved thanks to the memory engine inside the shielded enclave. In fact, the memory engine allows to read and write encrypted and authenticated data. When the IP (i.e., accelerator) produces data, it is sent to the shielded enclave. Inside the memory engine, the message signature is obtained with a Message Authentication Code (i.e., MAC). Then, the data is encrypted with a symmetrical encryption algorithm such as AES using $DataEncKey$. Finally, both

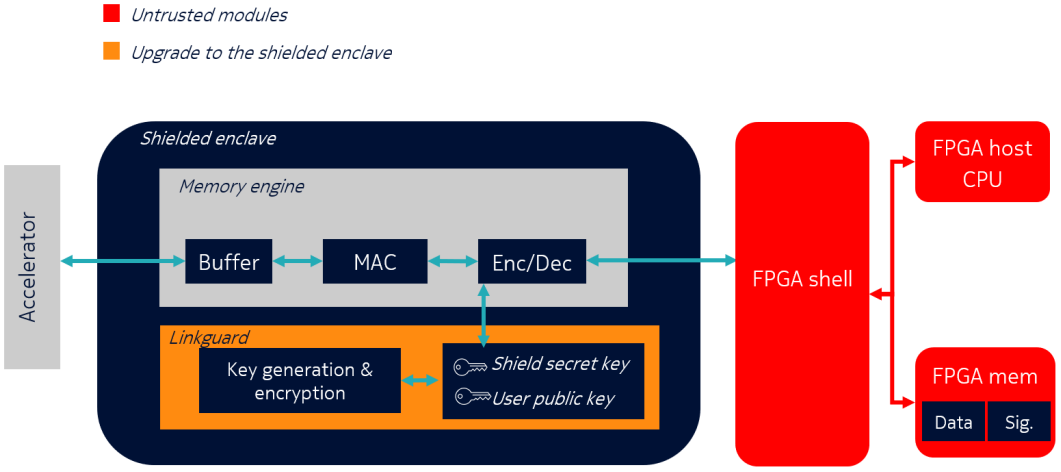


Figure 4-10 – Shield overview with the proposed upgrade

the encrypted data and the signature can be stored inside the memory. To consume data, the accelerator sends a read request through the shield for the FPGA shell for the message and its associated signature. The message is decrypted using AES algorithm and the received signature is verified inside the shield against the computed signature. If they match, the data is authenticated, remained confidential inside the device memory and its integrity is protected.

3.2 Threat model

Let's consider the scheme presented in Figure 2-13 and Figure 4-9 that is used in [7]. In this threat model, the attacker's objective is to break the integrity and the confidentiality of the data processed inside the FPGA. We assume that the adversary has physical access to the FPGA and any memory that the data owner has access can be compromised. In Figure 2-13 and Figure 4-9, red modules are untrusted. The shield logic and the security kernel are open source and are considered trusted. The Security Processor Block (i.e., SPB) runs the device firmware, initializes the FPGA alongside the security kernel and holds the device root-of-trust (i.e., keys embedded by the device manufacturer). The security of the SPB is in the CP interest because it holds critical security elements. The SPB does not add any complexity to the threat model, it is considered secure.

We assume that the CP and the IP vendor may be malicious. Thus, solutions, protocols and software provided by these entities may be inherently malicious. For example, the FPGA shell provides interfaces to the shield IP and it is deployed by the CP. We assume that the shell is able to log and store communications and

data belonging to the data owner. Although the CP and the IP vendor are both considered malicious, they do not cooperate to compromise the cloud user's data.

However, the adversary is not interested in Denial-of-Service (i.e., DoS) attacks and attacks that would damage the CP infrastructure. Some of them are bitstream level attacks [12], hardware Trojans [138] and multi-tenant attacks [73], [123]. These challenges are not addressed by our solution. We believe that existing bitstream verification tools [3], [11], [139] are already addressing this issue.

When compared to the literature, the scheme detailed in [7] is the most advanced solution to securely benefit from FPGA cloud acceleration while supporting multi-tenancy. However, this scheme has some security drawbacks.

The following section analyzes the vulnerabilities of the ShEF framework [7] in order to expose its security drawbacks.

3.3 Vulnerability analysis

First, let's consider the CP as malicious. Current CPs like Amazon AWS and Microsoft Azure have mandatory bitstream verification [3], [4]. During this verification, the CP is looking for malicious or harmful design patterns in the bitstream to protect the hardware and the cloud infrastructure. As the bitstream is not encrypted during the verification process, the CP can find the $ShieldEncKey_{priv}$ in the bitstream which was embedded by the IP vendor. Hence, the $DataEncKey$ can be retrieved by the CP. The latter can decrypt the load key sent by the data owner using the embedded $ShieldEncKey_{priv}$ found during IP verification. As a consequence, data transfers between the data owner and the FPGA would not be secure. Embedding sensitive keys inside the FPGA bitstream is not a secure way to provision keys for public FPGA clouds.

The IP vendor can also be considered malicious towards the data owner. As stated in Chapter 3.1, the $ShieldEncKey_{pub}$ and $ShieldEncKey_{priv}$ keys are generated by the IP vendor. If the load key is intercepted, the IP vendor could decrypt the load key using the $ShieldEncKey_{priv}$. To obtain the load key, the IP vendor can intercept the communications between the FPGA and the data owner because the latter must send the load key to the FPGA to set up the encrypted communication channel. Additionally, the IP vendor could implement malicious logic inside the accelerator to log data and retrieve the data encryption key that is generated by the data owner. The $BtstrmEncKey$ key is managed by the IP vendor, the latter is the certificate authority for the bitstream. The IP vendor can modify the integrity of the bitstream

by adding malicious elements such as hardware Trojans [140], or other malicious circuits [121], [124] to collect information on the data owner. This operation can be done on the initial bitstream given to the data owner but also dynamically during runtime. Furthermore, the IP vendor can communicate with the Security Kernel (i.e., SK) located inside the FPGA [7]. The attestation mechanisms and the communications are handled by the SK. Any potential vulnerability on FPGA reconfiguration would allow the IP vendor to reconfigure the FPGA by malicious bitstreams.

Using the presented scheme, the CP cannot be an IP vendor. The CP owns the cloud infrastructure and has privileged access to its resources. If the shield encryption key is generated by the CP, the load key of the data owner can be easily compromised. As a consequence, the communications between the FPGA and the data owner would not be secure. Additionally, the CP can attack the integrity of the bitstream and implement malicious design elements in the IP requested by the data owner. This can help to retrieve more data from the data owner.

The following section proposes an upgrade to the shielded enclave to enhance its security.

3.4 Proposed upgrade to the shielded enclave

Linkguard is proposed to address the vulnerability of the load key explained in Section 3.3. The data owner (i.e., cloud user) must generate encryption keys independently of any stakeholder to guarantee data confidentiality and key security. After the bitstream is configured inside the FPGA as described in Section 3.1, the encryption key inside the shield must be initialized before generating any data. Linkguard must generate a key thanks to the protocol described in Section 2 and send the generated Shield Secret Key (i.e., SSK) to the data owner (i.e., cloud user). After SSK verification, the shield is activated and it is able to process data. Figure 4-10 shows the upgrade brought to the shield in orange (i.e., Linkguard). Instead of storing a $DataEncKey$ inside the key store, we only need to store a user public key and the generated SSK by Linkguard. Because the SSK is generated by the FPGA and inside the shield, it is independent from the IP vendor and the CP. The data owner can create a communication channel and use the device memory by using the SSK.

Nevertheless, some threats still remain even after using Linkguard inside the shielded enclave. These threats are originating from the architecture proposed in [7]. In fact, the IP vendor tries to protect the confidentiality of his IP with various

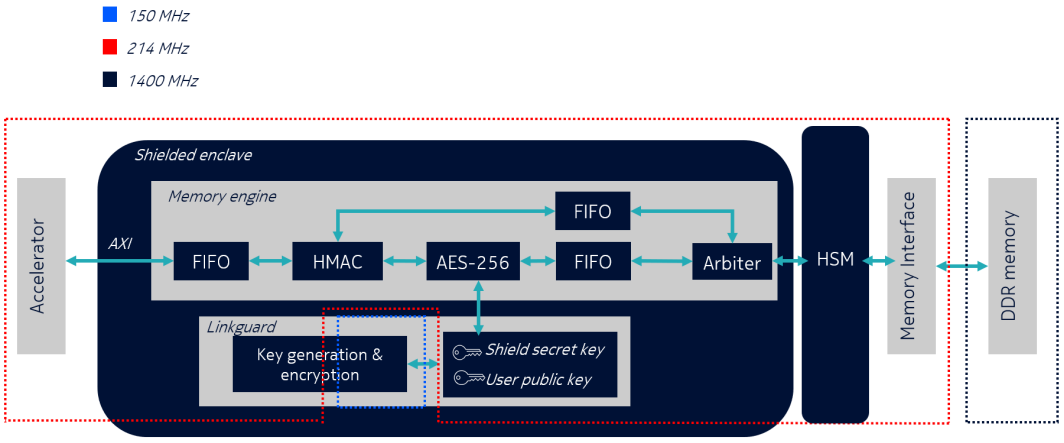


Figure 4-11 – All TokSek modules and their frequency

encryption techniques and protocol. Because of that, the data owner cannot know if the IP obtained by the IP vendor is safe or if it does contain any Trojans that can compromise his secure communication channel. This architecture requires that the data owner blindly trusts the IP vendor. In this situation, the user data is at stake. Nowadays, data is considered as one of the most valuable assets in the world [6]. It is not acceptable from the data owner perspective to have no guarantee of data confidentiality. The data owner could have valuable data like medical records, identity and faces to run machine learning applications. Such sensitive data must be protected to preserve the privacy of individuals or any commercial value that the data has.

In [7], the cloud FPGA access happens between the CP, the IP vendor and the data owner. Because this scheme does not guarantee data confidentiality for the data owner, we upgraded and adapted the use case of the shielded enclave for TokSek to achieve data confidentiality for the user. In TokSek, we have the CP, the TA and the cloud user that are present inside the FPGA access scheme. In this case, the cloud user develops the FPGA IP on his own independently from an IP vendor. However, as the shielded enclave proposed by [7] is open-source, the cloud user is responsible for implementing his IP with a shielded enclave that is upgraded with Linkguard. This way, the cloud user remains independent of any stakeholder for protecting his interests. Figure 4-11 is a high-level view of the upgraded shielded enclave and the logic modules of TokSek. Frequency regions are also illustrated and the figure shows that the shield and the HSM use a 214 MHz clock whereas Linkguard uses both 214 MHz and 150 MHz. The impact of having multiple frequency domains in TokSek is further analyzed in Section 4.

The current section presented a shielded enclave solution from the literature [7].

It is a solution that aims to protect the IP vendor bitstream and the data of the data owner. There are few security drawbacks that are identified inside the solution. This security issue renders the data of the cloud user vulnerable to the IP vendor in multiple ways as described in Section 3.3. Linkguard is proposed to address this issue. By generating encryption keys inside the FPGA, the vulnerability of the user data encryption key (e.g., load key as called in [7]) can be mitigated. By fixing this issue, the shielded enclave combined with Linkguard is added to TokSek to protect the data and the IP of the cloud user. The following section exposes the implementation results of the hardware modules described in this chapter. We analyze the resource utilization and the overall latency of each module.

4 Implementation and results

For this implementation, we used the same hardware target as the last chapter: AMD UltraScale+ ZCU 102 MPSoC. In this section, the resource utilization and the latency of each solution is exposed and compared with the literature.

One important aspect of this implementation is the design of some modules (e.g., RSA, HSM) using High-Level Synthesis (i.e., HLS) with the AMD Vitis development software. HLS is a design methodology that allows hardware descriptions using high-level programming languages like C, C++, or OpenCL, enabling a more abstract and fast way to design hardware circuits for FPGAs. The AMD HLS technology aims to accelerate the process of FPGA development by translating high-level software code into hardware circuits. The advantages of HLS include faster design generation, improved productivity, and the ability to leverage software developers knowledge for hardware design. This can potentially reduce time-to-market and make FPGA programming more accessible. However, drawbacks may include the challenge of optimizing complex algorithms for hardware efficiency, as automatic translation might not always yield the most optimized hardware implementation. Additionally, HLS tools may face limitations achieving the same level of performance as manually optimized hardware designs.

4.1 Resource utilization

4.1-1 HSM

Table 4-1 shows the logical resource utilization of various HSM components. It is important to recall that these modules are developed using high-level synthesis. The *auth_token* function is the most resource consuming element of the HSM due to the presence of cryptographic elements like a hash function (e.g., SHA-256). In fact, the hash function alone requires 5624 LUTs and 8519 FFs which is 45% of the LUTs and 47% of the FF usage of the *auth_token* module.

FPGA resources	LUT	FF	BRAM
auth_token	12550	18077	4
token_parse	6594	10474	3
allocate_mem	4260	5119	4
addr_translate	2294	3089	1
Linkguard*	22426	12782	0
Total	48124	49541	12
Percentage of ZCU 102	17.6%	9%	1.3%

Table 4-1 – Logical resource utilization for the components of the HSM developed using high-level-synthesis

We can compare the SHA-256 block developed with HLS and the one used inside the shielded enclave described in Section 3. The shield uses a SHA-256 block for the HMAC function. The hash function is developed using a hardware description language (e.g., Verilog). The RTL implementation of the SHA-256 module utilizes 2144 LUTs and 2095 FFs. The HLS version of the SHA-256 algorithm utilized 2.6 times more LUTs and 4.1 times more FFs. We can also similarly compare our *token_parse* function against the AMD HLS library. In fact, there is a JSON parser inside the AMD HLS library in AMD Vitis HLS. The latter has been implemented to be compared against our parser that is developed from scratch using HLS. The JSON parser of the AMD HLS library used 19057 LUTs 12535 FFs 22 BRAMs and 88 DSPs. This implementation has an excessive resource usage compared to the one we propose. In fact the parser from the library uses 2.9 times more LUTs, 1.2 times more FFs, 7.3 times more BRAMs than our parser. Additionally, our parser did not use any DSPs whereas the parser of the AMD library used 88 DSPs. Both the SHA modules and

FPGA resources	LUT	FF	BRAM	DSP
SHA-256 RTL	2144	2095	0	0
SHA-256 HLS	5624	8519	0	0
AMD HLS JSON parser	19057	12535	22	88
Proposed HLS JSON parser	6594	10474	3	0

Table 4-2 – Comparison between RTL and HLS modules developed for the HSM

the token parsers are compared in Table 4-2.

In total, one HSM uses less than 10% of the target device LUTs and less than 7% of the device register. The HSM is the foundation of the FPGA security and it enforces many mechanisms like access control, resource allocation, and token authentication.

It is important to note that Linkguard resource usage is not considered in this section because it is described in the next section.

4.1-2 Linkguard

The resource consumption of Linkguard is described in Table 4-3. The cryptographic library of AMD Vitis is used to implement the RSA encryption with HLS. When we compare the proposed RSA implementation against the literature, the logical resource consumption achieved in Linkguard is incredibly high. In [141], an RSA cryptosystem has been implemented using 16952 LUTs and 2058 registers. Their implementation is using an optimized serial Montgomery modular multiplication that is developed using a Hardware Description Language (i.e., HDL). The RSA implementation in [141] used 82% less LUTs and 98% less registers compared to our RSA implementation using HLS. The difference of resource consumption between our RSA implementation and [141] is high and it effectively showcases the drawbacks of high level synthesis for FPGA logic development. Naturally, an optimized RSA implementation is preferable to be used in Linkguard. By reducing its footprint, Linkguard can be more suitable for FPGA cloud multi-tenancy where logical resources are the most valuable.

The following section gives the resource utilization for the upgraded shielded enclave.

Logical resource consumption for Linkguard			
FPGA resources	LUT	FF	BRAM
TRNG	231	545	0
HMAC	3062	2257	0
NIST DRBG	2181	7922	0
HLS RSA encryption	94430	97418	61
RSA [141]	16952	2058	N/A
Total w/ HLS RSA	99904	108142	61
Total w/ RSA [141]	22426	12782	0
Percentage of ZCU 102	8.2%	2.3%	6.7%

Table 4-3 – Resource consumption of the upgraded shielded enclave on the ZCU 102

4.1-3 Upgraded shielded enclave

As shown in Figure 3, the shield needs a MAC algorithm and a symmetrical encryption algorithm. For the MAC algorithm a SHA256-based HMAC is used to guarantee data integrity and confidentiality at the same time. For symmetrical encryption, AES-256 is implemented. These two cryptographic blocks are used in an encrypt-then-mac method scheme. This method is considered the safest option [48] as stated in Chapter II Section 3.

The logical resource consumption of the shielded enclave using the ZCU 102 is shown in Table 4-4. The maximum clock speed is set at 214 MHz in this implementation. The critical path of this design lies in the HMAC module. To limit the complexity of this design and the clock tree, we did not seek to optimize to a great extent. This design has three different clock regions: 150 MHz, 214 MHz and 1400 MHz. The slowest clock is required for the RSA encryption in Linkguard as described in Section 2. The 214 MHz clock is the base clock of the TokSek design and it is the limit of the HMAC module. Lastly, the fastest clock is reserved for the DDR memory. Having too many clocks increases the implementation complexity of an FPGA design by creating multiple clock domain crossings (i.e., CDC). When signals need to be transferred between these domains, issues such as metastability and data corruption can arise due to the mismatch in clock frequencies and phases. The purpose of CDC is to manage and handle the challenges associated with transferring signals between these clock domains. The existence of CDC mechanisms in FPGA design

helps ensure proper data synchronization and integrity when crossing clock boundaries. Each time that two modules running at different speeds must communicate, additional logic must be created. Among these, we can find components like FIFOs to transfer data between clock domains.

FPGA resources	LUT	FF	BRAM
Control and interfaces	1486	1367	3
Write encryptor	5229	4468	0
Read decryptor	5234	4470	0
Linkguard*	22426	12782	0
Total	34375	23087	3
Percentage of ZCU 102	12.5%	4.2%	0.3%

* The optimized version of Linkguard with the RSA implementation described in [141]

Table 4-4 – Resource consumption of the upgraded shielded enclave on the ZCU 102

FPGA resources	LUT	FF
AES-256	2167	2209
SHA-256-HMAC	3062	2257

Table 4-5 – Resource consumption of the cryptographic modules implemented inside the shield

The logical resource consumption of the shield is reduced due to its modularity. Achieving small overhead on logical resources is an important requirement for FPGA acceleration solutions and it is of the utmost concern for multi-tenant FPGA clouds. In fact, each user must implement a shielded enclave to protect his own data. The overhead of all security solutions deployed in FPGA logic must be small enough to leave logical resources for actual acceleration. Because each user gets a finite amount of logical resources inside an FPGA cloud, the security mechanisms must not prevent users from deploying acceleration solution. However, after adding Linkguard to the shielded enclave, the resource utilization is increased. The upgraded shield uses 12.5% of the total available LUTs and 4.2% of the registers. Such resource

utilization is adapted for multi-tenant FPGA clouds because the overhead is low enough to place multiple tenants per FPGA.

We can illustrate this claim with a practical example. Let's consider three tenants using a shielded enclave with one neural network accelerator each [66]. The resource utilization of the neural network proposed in [66] is 38899 LUTs, 40534 FFs, 3 BRAMs. This respectively represents 14.2%, 6.8% and 0.3% of device LUTs, FFs and BRAMs . By including the resource utilization of HSM, it is possible to securely share the FPGA among 3 users. The total LUTs utilization of the AMD ZCU 102 MPSoC would be $3 \times 14.2 + 3 \times 12.5 + 9.4 = 94\%$. The FF utilization would be $3 \times 6.8 + 3 \times 2.3 + 6.7 = 34\%$ and the BRAMs utilization would reach $3 \times 0.3 + 1.3 = 2.2\%$. With this use case, the LUTs utilization would be nearly maximized with three users implementing the same neural network accelerator with a shield. With high-end FPGA devices like AMD Alveo and Intel Stratix, the resource utilization would be lower and more tenants could share the same FPGA.

After exposing the resource utilization for the proposed framework and the possibility of secure multi-tenancy, we detail the timing aspect of TokSek components in the next section.

4.2 Latency

4.2-1 HSM

In terms of latency, each module has been tested with various use cases to analyze the behavior of the design. Table 4-6 shows the latencies for the *auth_token* function under four different use cases like a modified token with bad integrity, a valid token, a token signed with a wrong fss_j, and a token signed with a wrong hash algorithm. The latency of the modified token and the valid token are similar with 0.205 ms and 0.209 ms respectively. For the modified token, the content has been altered without forging a new signature because the fss_j key is kept secret by the TA. However, on the bad fss_j use case, the wrong secret key is used to sign the token and the algorithm took 0.232 ms to refuse the validity of the token. The last test case is the usage of a wrong hashing algorithm. In fact, when the algorithm specified in the token header is different than the hashing algorithm used for verification, the token signature cannot be verified. The token has been refused in 0.167 ms for this case.

After the token validation, the user policies must be initialized. The token is parsed, the payload is retrieved and the memory is allocated. To test the memory allocation algorithm, we start from an empty memory and we allocate 128 bits for

	Hardware execution	Token valid
Modified token	0.205 ms	×
Valid token	0.209 ms	✓
Bad fss_j	0.232 ms	×
Wrong hashing algorithm	0.167 ms	×

Table 4-6 – Latency of the *auth_token* module

three users. Figure 4-12 shows the allocation process for one specific scenario. The random start addresses are noted on the left and the memory regions are noted in the table on the right-hand side. It is important to note that the data stored in the table represents the memory block information $mbi_{i,n}$ stored inside the policy store ps_j.

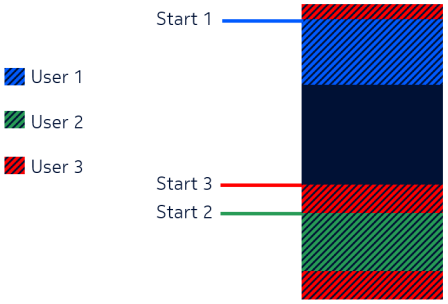
Table 4-7 shows the latency incurred by the *allocate_mem* function. We observe that the order of magnitude for the memory allocation is 10^{-6}s. The latency is negligible compared to the token authentication. Additionally if we take into account the token authentication latency and the memory allocation latency, all the user policies can be initialized in 0.211 ms for the worst case scenario according to our measures.

	Hardware execution	$frag$ count
User 1 memory allocation	1.09 µs	1
User 2 memory allocation	1.13 µs	1
User 3 memory allocation	2 µs	3

Table 4-7 – Latency of the *allocate_mem* module

Dynamic memory allocation is a highly discussed subject in the literature because it can impact the overall system performance and efficiency [142], [143], [144]. The memory allocator in TokSek uses a sliding window approach that is described in Section 1.2-2. The focus was on achieving a compact solution to minimize the logic usage because the memory allocator is only used once after token authentication. For this reason we did not use any performance optimization techniques to achieve a very low latency solution. When we compare the resource usage, our memory allocator utilizes 4260 LUTs of the Zynq ZCU 102 whereas solution [142] use between 1415 and 2171 LUTs of the AMD Zynq 7000 SoC. Our allocator uses between two and three times more LUTs than [142]. Moreover, their allocator has a latency of 8×10^{-8}

s whereas our allocator has a latency of 2×10^{-6} on the worst case, which is 25 times more. This difference shows that the current implementation of TokSek can be optimized to achieve better performance results. Work [143] is another memory allocator implemented on Zynq 7000 SoC. It used 12236 LUTs and has an allocation latency of 1.9×10^{-7} s. Compared to our allocator, [143] is 10.5 times faster but it uses 2.9 times more LUTs. The proposed allocator is more efficient for resource utilization and this is very important for a multi-tenant context. The memory allocator is only used one time after user authentication. Hence, minimizing resource utilization is more important than minimizing latency for our case.



User ID	Memory region	Allocated size
User 3	0x00 – 0x0A	0x0A
User 1	0x0B – 0x8B	0x80
User 3	0x108 – 0x113	0x0B
User 2	0x114 – 0x194	0x80
User 3	0x195 – 0x200	0x6B
Total RAM	0x00 – 0x200	75.6% usage

Figure 4-12 – Example of memory allocation procedure with fragmentation

After memory allocation, we can observe the latency of the *addr_translate* function where a user u_i sends a virtual address. The HSM must find in the $mbi(i, n)$ the corresponding physical address inside the policy store ps_j to fulfill the user memory request. Table 4-8 shows the latency for the memory translation function. For each user, we request the translation of the virtual address 0x05. We observe that the latency for address translation has a relatively small overhead around 90 µs. If we consider an AXI write request with 16 bursts of 64 bits of data (1 MB total) the overhead for address translation is between 1.44 ms and 1.55 ms. This overhead can be further minimized with optimization techniques like pipelining because the maximum frequency of the *addr_translate* module is 328 MHz which is more than the Shield-HSM communication clock that runs at 214 MHz.

When compared against other address translation mechanisms in the literature, our solution has more latency. In [145], a memory translation module is implemented on an Intel Stratix 4 FPGA and they compared the performances against a softcore NIOS II CPU. As a baseline, the NIOS CPU achieves address translation in 1.9 µs which is similar to our solution on the worst case but we are 1.9 times faster on our best measured case. However, a Translation Look-aside Buffer (i.e., TLB) has been implemented in [145]. It is a hardware cache that stores the recent translations of

virtual memory addresses to physical memory addresses. Their TLB has been implemented using 11450 LUTs and 8303 FFs which corresponds to 5 times more LUTs and 2.7 times more FFs than our translation unit. Yet, they achieve a translation latency of 124 ns which is 8.1 times faster than our solution.

	Hardware execution	*frag* count
User 1 address translation	0.090 ms	1
User 2 address translation	0.095 ms	1
User 3 address translation	0.097 ms	3

Table 4-8 – Latency of the *addr_translate* module for the translation of the address 0x05

4.2-2 Linkguard

Table 4-9 shows the latency of individual blocks that are present in Linkguard. The majority of the overhead comes from the TRNG and the RSA encryption. Our RSA implementation may be excessively resource consuming but if we compare the latency with [141], our RSA implementation with HLS is slightly faster. In [141], the encryption latency is about 2.2 ms whereas we achieved 1.59 ms which is 22% faster. We can also compare our TERO-TRNG implementation with [131]. In fact, our TRNG generated 440 random bits in 5.26 ms which gives us a bitrate of 0.084 Mb/s. In [131], the TERO-TRNG is reported to have a bitrate between 0.625-1 Mb/s according to the FPGA device. The bitrate achieved in this implementation is 7.5 times slower than [131].

Achievements	Computing time (ms)
TRNG key generation	5.26 ms
NIST DRBG	648×10^{-6} ms
HMAC signature	1.14×10^{-3} ms
RSA encryption	1.59 ms
Total	6.85 ms

Table 4-9 – Timing results of Linkguard

4.2-3 Upgraded shielded enclave

Moreover, the impact on memory bandwidth is shown in Table 4-10. We did not reach maximum memory performance for few reasons. The memory interface operates at 214 MHz whereas the DDR4 memory itself runs at 1200 MHz. The acronym DDR means "Double Data Rate" because the memory samples on rising and falling edges. Consequently, a DDR4 memory clocked at 1200 MHz is working at double speed an it is noted at DDR4-2400. This means a DDR4-2400 memory is capable of executing 2400 megatransfers per second. The maximum theoretical bandwidth of a DDR memory module is calculated based on its data rate and bus width. In this case, the data rate is 2400 megatransfers per second whereas the shield and the memory interface have a 64-bit wide AXI bus. Mathematically, the maximum theoretical bandwidth is expressed as:

$$theoretical\ bandwidth = \frac{2400 \times 64}{8 \times 10^3} = 19.2\ GB/s \tag{4-12}$$

However, the shield and the memory interface use a clock that runs at 214 MHz due to design constraints. Consequently, this implementation cannot reach the theoretical maximum memory speed because the shield cannot produce data fast enough. The maximum bandwidth that this implementation can achieve is expressed as follows:

$$achievable\ bandwidth = \frac{214 \times 64}{8 \times 10^3} = 1.7\ GB/s \tag{4-13}$$

There is a 5.6 times factor between the theoretical and achievable bandwidth respectively expressed in Equation 4-12 and 4-13.

Memory speed comparison with the shield and the DDR	
Read/Write Operations	6.8 Gbit/s (0.85 GB/s)
Theoretical DDR4 2400 MHz	19.2 GB/s

Table 4-10 – Memory speeds with the shield

The experimental memory bandwidth measured when using the shield is 0.85 GB/s. This result is 2 times slower than the maximum achievable bandwidth. The reason for such a slowdown is the bottleneck inside the HMAC module that prevents the design from reaching faster speeds. No optimization and implementation

techniques like pipelining and parallelism have been studied in order to reach higher performances because it is not the objective of this study.

5 Summary

Section 1 introduced the last component of TokSek: the HSM. It is responsible for crucial security functions to ensure a secure multi-tenant FPGA environment within the cloud. It enforces access control policies, allocates resources, and provides isolation between entities. The HSM components are developed using High-Level Synthesis (i.e., HLS), allowing faster development with some trade-offs in resource utilization. The HSM introduces mechanisms such as a hardware token authentication, a hardware resource allocation by parsing a JWT access token and a dynamic memory allocation mechanism. In fact, the latter allocates user memory according to the user token content. The memory allocator handles fragmentation and uses random starting address to allocate the first memory address. This helps to mitigate memory attacks like Rowhammer and aims to keep the user memory location secret. Access control functions like *check_policy* and *addr_translate* enforce security policies and translate virtual addresses into physical addresses, adding an extra layer of abstraction. In fact, each virtual address received by a user FPGA logic is interpreted differently thanks to the policy store ps_j containing memory block information $mbi(i, n)$.

In this chapter, Section 2 introduces Linkguard, a zero-trust key generation protocol for establishing secure communication channels, particularly within untrusted cloud environments. It highlights the role of Linkguard in addressing privacy concerns associated with third-party cloud providers. This solution uses a TRNG to generate a random seed for a DRBG that produces a session key. The latter is protected with asymmetric cryptography and a MAC algorithm to ensure confidentiality and integrity. Then, a practical use case for Linkguard applied to O-RAN is detailed to show the capabilities of this solution. In fact, Linkguard addresses multiple security vulnerabilities that are identified by the O-RAN security work group.

Section 3 describes a shielded enclave solution sourced from the literature, where security concerns related to the protection of IP vendor bitstream and user data are identified. The shielded enclave allows an accelerator to read and write encrypted data while ensuring data integrity. Linkguard is proposed as a solution to mitigate vulnerabilities through the internal generation of encryption keys within the FPGA. When integrated into TokSek, the upgraded shielded enclave offers a comprehensive

approach to safeguarding both data and IP in cloud environments.

Section 4 exposes the implementation results in terms of resource utilization and latency for the HSM, Linkguard and the upgraded shielded enclave. Inside the HSM, there are four important functions: *auth_token*, *token_parse*, *allocate_mem* and *addr_translate*. The resource utilization of the HSM is 9.4% of device LUT and 6.7% of device FF. The implementation shows 0.211 ms of hardware execution for token authentication, parsing and memory allocation. This is an acceptable overhead for a one time execution that initializes the user acceleration context inside the FPGA. Additionally, the implementation of Linkguard shows that it is possible to generate a session key securely and independently from the CP in 6.85 ms. However, the proposed implementation of the HLS RSA module has a high logical resource consumption, which may impact efficiency in multi-tenant scenarios. This issue is implementation specific and rooted in HLS that offers a fast but unoptimized implementation. Other efficient RSA implementations exist and can be used in another Linkguard implementation. Then the shielded enclave in [7] is analyzed and some security vulnerabilities are exposed. The shield module aims to protect a user accelerator inside the FPGA against malicious I/O and protect the confidentiality and integrity of user data. An upgrade of the shielded enclave module with Linkguard is proposed to address vulnerabilities. The upgraded shielded enclave is implemented and utilizes 10.5% of device LUT and 2.3% of device FF. In terms of memory bandwidth, a 2 times slowdown is observed compared to the achievable memory bandwidth. This is an acceptable overhead compared to the security level this solution brings. Some of the hardware modules in TokSek are not as optimized as the ones from the literature. However, this work describes a complete security framework for token-based multi-tenant FPGA cloud acceleration.

Chapter V

Conclusion

The primary objective of this thesis is to provide a secure multi-tenant FPGA cloud framework that protects user data confidentiality and acceleration environment. The secondary objective of this thesis is to increase the scalability, to minimize the overhead and the complexity of this framework. A good quality of service must be provided without degrading overall system performance, security and resource consumption. Because FPGA logic resources are valuable, the logic utilization for architecture and security must be constrained to leave as much logic resources as possible for acceleration purposes.

The content of the chapters in this thesis are summarized below and future works are described.

1 Summary

Chapter II offers an overview of multi-tenant FPGA cloud computing and its security needs. It introduces current FPGA cloud architectures for cloud computing by IBM and Intel while discussing cloud service providers like Amazon, Alibaba, and Huawei. The chapter explores Trusted Execution Environments (i.e., TEEs) for FPGA, detailing technologies like Arm TrustZone, Intel SGX, and Keystone, all aiming to create secure execution environments. Existing TEE vulnerabilities are exposed and some mitigation techniques are described. Security is further enhanced through authentication techniques, including HMAC and Authenticated Encryption (i.e., AE) solutions. FPGA-specific authentication methods, such as using Physical Unclonable Function (i.e., PUF) and using a TA are also discussed. The concept of FPGA multi-tenancy is introduced, employing virtualization tools to establish isolated environments for multiple users. The chapter addresses hardware-based vulnerabilities in multi-tenancy and proposes mitigation techniques. Lastly, it emphasizes the importance of FPGA access control in multi-tenant environments for secure resource sharing among FPGA tenants. We showed that access control mechanisms

are not popular in FPGA acceleration because they are only necessary for multi-tenant environment. No cloud provider proposes multi-tenancy for FPGA clouds at the time of writing, thus, no access control mechanisms exist for current FPGA cloud deployments.

Chapter III dives into TokSek: a token-based multi-tenant FPGA cloud security framework. In Section 1, we set the foundation of the framework by defining potential threats and creating a formal set of rules for TokSek. The threat model outlines what attackers, whether inside or outside the FPGA, might aim to do. The formalization is a theoretical guide for TokSek mechanisms.

In Section 2, we introduce TokSek, talking about how we have adapted OAuth 2 for FPGA clouds and explaining the token system. Section 2.2 gets into the details of the TokSek protocol, showing how different parts communicate and handle tasks. The architecture of TokSek has four major entities, the CP, the TA, the user and the FPGA. The user requests cloud resources from the CP. The latter needs the TA for FPGA access token generation. The user retrieves the access token from the TA and connects to the FPGA. For each communication, entities mutually authenticate each other and sensitive data is signed using shared keys. The access token is one sensitive data that is signed by the TA using a shared secret with the FPGA. Section 2.3 suggests an extension of OAuth 2, letting authorized users to share access to their allocated resources with other users. An example involving O-RAN and 5G technologies is detailed. In such use cases, a mobile network provider can share its on-premise and third-party infrastructure with other network providers. Our results indicate that using tokens for FPGA access has minimal latency. In fact, a cloud user can obtain a token in about 183 ms and access the FPGA in 116 ms. In total, a user can get their access token, connect and reconfigure the FPGA in 503 ms. Compared to [63], our framework is 37.8% faster for FPGA access and bitstream configuration. It is important to recall that the target FPGA SoC is a standalone device in this implementation. A lightweight Flask server has been implemented inside an embedded linux for user connection. As a result we showed that the FPGA Arm CPU is 3x slower than a regular laptop's virtual machine server during mutual TLS authentication.

In Chapter IV, the hardware components of TokSek are described. In Section 1, we introduce the last piece of TokSek: the HSM. It's in charge of vital security tasks to ensure a safe multi-tenant FPGA environment in the cloud. The HSM enforces access control policies, assigns resources, and creates isolation between different entities. High-Level Synthesis (HLS) is used for faster development, even though it

comes with some trade-offs in resource use. The HSM introduces mechanisms like hardware token authentication, hardware resource allocation using a JWT access token, and dynamic memory allocation. This allocator manages fragmentation and keeps the first memory address secret to prevent memory attacks like Rowhammer. Access control functions, such as *check_policy* and *addr_translate*, enforce security policies and translate virtual addresses into physical addresses, adding an extra layer of security. The implementation shows 0.211 ms of hardware execution for token authentication and memory allocation, which is an acceptable overhead for a one-time execution. Moreover, in Chapter III, the user would achieve a TLS handshake in 101 ms and the user would be authorized to use the FPGA in 15ms. With the hardware implementation, the HSM initializes a user in 0.211 ms while providing access control mechanisms. It is a huge gain in latency compared to the software approach while providing more functions like memory allocation. Regarding resource utilization, the lack of efficiency is also observed with the JSON parser used inside the HSM. In fact, the JSON parser of the AMD Vitis HLS library used 2.9 times more LUTs and 7.3 times more FF compared to the HLS JSON parser proposed in this thesis.

Section 2 introduces Linkguard, a security protocol for creating safe communication channels, especially in untrusted cloud settings. It's designed to address privacy concerns when using third-party cloud provider resources. Linkguard allows to generate an encryption key inside the FPGA logic using a TRNG according to NIST recommendations [132]. The generated key is encrypted by the user's public key with RSA encryption. The proposed implementation of Linkguard can generate an encryption key in 6.85 ms which is a relatively small latency for a one time operation.

In Section 3, we discuss a shielded enclave solution from the literature. It tackles security issues related to protecting IP vendor bitstreams and user data. This shielded enclave encrypts produced data and decrypts consumed data for the acceleration IP. It also ensures data integrity with the HMAC algorithm by writing the data and its signature on each memory operation. Linkguard is suggested to improve vulnerabilities by generating encryption keys internally within the FPGA. When added to TokSek, this upgraded shielded enclave is a comprehensive way to protect both data and IP in cloud environments, with only a two times slowdown compared to memory bandwidth.

In this thesis, previously set objectives are achieved. TokSek is a token-based multi-tenant security framework that provides end-to-end security for FPGA cloud users. This solution reinforces data confidentiality and provides a secure multi-tenant FPGA environment. At the time of writing, TokSek is the first solution that provides

a complete hardware-based security framework with features like memory allocation with fragmentation, address virtualization, and token-based operations. Linkguard is another important security element of TokSek. This patented solution allows to create isolation between the CP and the cloud user by generating encryption keys and setting up an encrypted communication channel between the user FPGA logic and the cloud user.

2 Future works

There are still some areas to investigate in this thesis to obtain an efficient multi-tenant FPGA cloud framework. In fact, one of the most important aspects is the area usage for the FPGA logic. In this thesis the HSM is developed using AMD Vitis HLS. It allowed to build a proof of concept for this framework to show its potential and feasibility. However, when compared to the literature, our solution uses too many resources. By developing the HSM using hardware description languages such as Verilog and VHDL, it is possible to obtain and optimized solution. Logical resource usage is of the utmost importance for multi-tenant FPGA cloud because the logic resources are used for acceleration purposes.

Another area of investigation is the deployment of TokSek in a state-of-the-art cloud environment. In fact, current FPGA clouds can be deployed using virtualization tools like Docker and Kubernetes as described in Chapter II Section 5.2. Figure 5-1 proposes a potential architecture of deployment using Kubernetes. On the first step, the resource requirement of a user instance is sent to the Kubernetes master to seek for an available FPGA device. On the second step, the FPGAs answer to the Kubernetes master to report on resource availability. At this stage, the TokSek framework can begin, and the CP can send resource requirements and access rules to the TA for access token creation. The cloud user must obtain an access token in order to proceed with step 3 and achieve mutual TLS authentication with the authentication pod and make a deployment request. The authentication pod fetches the user application and sends it to the HSM on step four. The user application is verified on step five thanks to the HSM then deployed in its allocated pod on step six. On step seven, the user application sends its access token and the bitstream to the HSM for verification and upon successful verification with the *auth_token* function, the user's authorizations are initialized in the FPGA logic thanks to the memory allocator and the policy store. On the same step, the user's acceleration slot (e.g., reconfigurable region $rr_{i,j}$) is reconfigured. On step nine, the user acceleration

environment is ready to use.

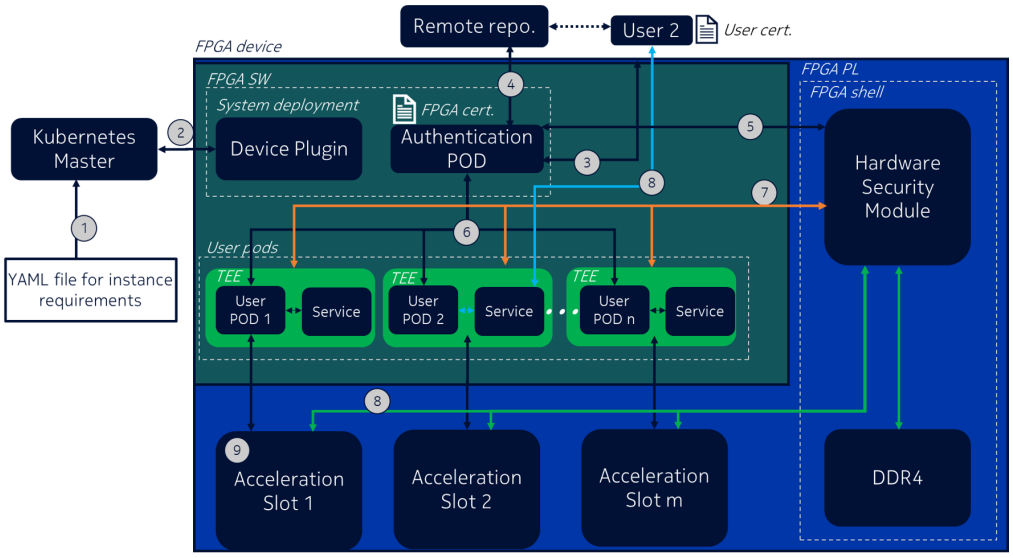


Figure 5-1 – Example of cloud deployment using Kubernetes

Such a deployment will allow to further experiment with the framework under a real use case scenario to obtain more accurate experimental performance results. Consequently, any potential bottlenecks or drawbacks can be identified and fixed. Observing impact of the architecture is important too. In the implementation proposed in this thesis, an FPGA SoC is used. According to experimental results, the SoC CPU is 3 times slower than a general purpose CPU for mutual TLS authentication. While our current SoC architecture makes each FPGA independent from a host, a significant amount of performance loss occurs in the functions inside the processing system. Comparing the FPGA SoC architecture against the more standard FPGA-host architecture is another interesting area to investigate. User application pods shown in Figure 5-1 can be deployed inside a general purpose server but in the context of multi-tenancy, this can be suboptimal. In fact, the FPGA can be connected to a host CPU using a PCI interface. If all tenants communicate at the same time with their reconfigurable region, the PCI port can be a bottleneck and degrade system performance. With a different architecture, new security challenges may appear. They need to be studied and addressed to provide a secure and privacy preserving acceleration environment.

Additionally, quantum computing is one major concern for cloud security as it will render most of the cryptosystems obsolete [146]. Asymmetric cryptography is the most impacted technology as quantum technology will most likely break RSA-2048 by 2031 [147]. Consequently, post-quantum cryptography (e.g., fully homomorphic encryption) must be leveraged to protect the cloud computing environment against

this threat [148] [149]. Current TEEs are also under the threat of quantum computing and post-quantum TEEs must be developed to be resilient against this technology. The threat of quantum computing must not be underestimated. Therefore, the current security architecture and cryptosystems deployed in TokSek must be analyzed to identify any potential vulnerabilities in order to mitigate them.

BIBLIOGRAPHY

[1] V. Ziegler, P. Schneider, H. Viswanathan, M. Montag, S. Kanugovi, and A. Rezaki, "Security and trust in the 6g era," *IEEE Access*, p. 1, 2021.

[2] F. Turan, S. S. Roy, and I. Verbauwhede, "Heaws: An accelerator for homomorphic encryption on the amazon aws fpga," *IEEE Transactions on Computers*, vol. 69, pp. 1185–1196, Aug. 2020.

[3] "Official repository of the aws ec2 fpga hardware and software development kit." `https://github.com/aws/aws-fpga`.

[4] "Microsoft azure documentation for fpga optimized virtual machine sizes,"

[5] "Alibaba fpga cloud documentation." `https://www.alibabacloud.com/help/en/fpga-based-ecs-instance`.

[6] P. G. Leonard, "Is data your most valuable asset that you never owned?," *IRPN: Governance (Sub-Topic)*, 2018.

[7] M. Zhao, M. Gao, and C. Kozyrakis, "ShEF: shielded enclaves for cloud FPGAs," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, feb 2022.

[8] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. Gatlin, M. Ghandi, and D. Burger, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, pp. 8–20, 03 2018.

[9] A. M. Caulfield and et al., "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–13, Dec. 2016.

[10] B. Ringlein, F. Abel, A. Ditter, B. Weiss, C. Hagleitner, and D. Fey, "System architecture for network-attached fpgas in the cloud using partial reconfiguration," in *2019 29th International Conference on Field Programmable Logic and Applications*, pp. 293–300, Aug. 2019.

[11] T. La, K. Mätas, N. Grunchevski, K. Pham, and D. Koch, "Fpgadefender: Malicious self-oscillator scanning for xilinx ultrascale+ fpgas," *ACM*

Transactions on Reconfigurable Technology and Systems, vol. 13, pp. 34:1–34:20, May 2020.

[12] R. Chakraborty, I. Saha, A. Palchaudhuri, and G. Naik, "Hardware trojan insertion by direct modification of fpga configuration bitstream," *Design & Test, IEEE*, vol. 30, pp. 45–54, 2013.

[13] "Microsoft azure documentation for cloud fpga attestation mechanism." `https://learn.microsoft.com/en-us/azure/virtual-machines/field-programmable-gate-arrays-attestation`.

[14] V. Costan and S. Devadas, "Intel sgx explained." Cryptology ePrint Archive, Paper 2016/086, 2016. `https://eprint.iacr.org/2016/086`.

[15] "Arm® trustzone technology for the armv8-m architecture." `https://developer.arm.com/documentation/100690/0201/`, 2019.

[16] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[17] "Intel stratix 10 product description." `https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10.html`.

[18] "Amd xilinx alveo boards." `https://www.xilinx.com/products/boards-and-kits/alveo.html`.

[19] "Amd xilinx soc platforms." `https://www.xilinx.com/products/silicon-devices/soc.html`.

[20] "Risc-v keystone enclave documentation." `https://www.xilinx.com/products/boards-and-kits/alveo.html`.

[21] "Risc-v rocket." `https://github.com/chipsalliance/rocket-chip/`.

[22] "Risc-v boom." `https://github.com/riscv-boom/riscv-boom/`.

[23] "Risc-v cva6." `https://github.com/openhwgroup/cva6/`.

[24] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[25] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International*

Workshop on Hardware and Architectural Support for Security and Privacy, HASP '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[26] E. M. Benhani, L. Bossuet, and A. Aubert, "The security of arm trustzone in a fpga-based soc," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1238–1248, 2019.

[27] Y. Ma, Q. Zhang, S. Zhao, G. Wang, X. Li, and Z. Shi, "Formal verification of memory isolation for the trustzone-based tee," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 149–158, 2020.

[28] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Comput. Surv.*, vol. 51, jan 2019.

[29] S. Pinto and C. Garlati, "Multi zone security for arm cortex-m devices," 02 2020.

[30] K. Xia, Y. Luo, X. Xu, and S. Wei, "Sgx-fpga: Trusted execution environment for cpu-fpga heterogeneous architecture," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 301–306, 2021.

[31] A. Nilsson, P. K. Bideh, and J. Brorsson, "A survey of published attacks on intel sgx," tech. rep., CoRR, abs/2006.13598, 2020.

[32] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 142–157, 2019.

[33] M. A. Mukhtar, M. K. Bhatti, and G. Gogniat, "Architectures for security: A comparative analysis of hardware security features in intel sgx and arm trustzone," in *2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE)*, pp. 299–304, 2019.

[34] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 339–354, 2021.

[35] Z. KOU, S. Sinha, W. HE, and W. ZHANG, "Cache side-channel attacks and defenses of the sliding window algorithm in tees," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2023.

[36] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 2, pp. 171–191, 2018.

[37] M. Gross, N. Jacob, A. Zankl, *et al.*, "Breaking trustzone memory isolation and secure boot through malicious hardware on a modern fpga-soc," *Journal of Cryptographic Engineering*, vol. 12, pp. 181–196, 2022.

[38] E. M. Benhani, L. Bossuet, and A. Aubert, "The security of arm trustzone in a fpga-based soc," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1238–1248, 2019.

[39] M. Gross, N. Jacob, A. Zankl, and G. Sigl, "Breaking trustzone memory isolation through malicious hardware on a modern fpga-soc," in *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, pp. 3–12, Association for Computing Machinery, 2019.

[40] M. Ghaniyoun, K. Barber, Y. Xiao, Y. Zhang, and R. Teodorescu, "Teesec: Pre-silicon vulnerability discovery for trusted execution environments," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[41] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," May 2020.

[42] Y. Xu, Z. Yu, D. Tang, G. Chen, L. Chen, L. Gou, Y. Jin, Q. Li, X. Li, Z. Li, J. Lin, T. Liu, Z. Liu, J. Tan, H. Wang, H. Wang, K. Wang, C. Zhang, F. Zhang, L. Zhang, Z. Zhang, Y. Zhao, Y. Zhou, Y. Zhou, J. Zou, Y. Cai, D. Huan, Z. Li, J. Zhao, Z. Chen, W. He, Q. Quan, X. Liu, S. Wang, K. Shi, N. Sun, and Y. Bao, "Towards Developing High Performance RISC-V Processors Using Agile Methodology," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1178–1199, 2022.

[43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," 2018.

[44] M. Parelkar, "Fpga security-bitstream authentication," technical report, George Mason University, 2004.

[45] S. C. Ramanna and P. Sarkar, "On quantifying the resistance of concrete hash functions to generic multicollision attacks," *IEEE Transactions on Information Theory*, vol. 57, pp. 4798–4816, July 2011.

[46] K. M. Abdellatif, R. Chotin-Avot, and H. Mehrez, "Protecting fpga bitstreams using authenticated encryption," in *2013 IEEE 11th International New Circuits and Systems Conference (NEWCAS)*, pp. 1–4, 2013.

[47] D. Maimut and R. Reyhanitabar, "Authenticated encryption: Toward next-generation algorithms," *IEEE Security & Privacy*, vol. 12, pp. 70–72, Mar.-Apr. 2014.

[48] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," *J Cryptol*, vol. 21, p. 469–491, 2008.

[49] E. Rescorla, "The transport layer security (tls) protocol version 1.3," tech. rep., RFC 8446, August 2018.

[50] R. Perlman, "An overview of pki trust models," *IEEE Network*, vol. 13, pp. 38–43, Nov.-Dec. 1999.

[51] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas, "Internet x.509 public key infrastructure: Certification path building," tech. rep., RFC 4158, September 2005.

[52] H. Englund and N. Lindskog, "Secure acceleration on cloud-based fpgas – fpga enclaves," *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 119–122, 2020.

[53] K. Eguro and R. Venkatesan, "Fpgas for trusted cloud computing," in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications*, pp. 63–70, 2012.

[54] S. Cantor, J. Kemp, R. Philpott, and E. Maler, "Assertions and protocols for the oasis security assertion markup language (saml) v2.0." OASIS Standard saml-core-2.0-os, March 2005.

[55] D. Hardt, "The oauth 2.0 authorization framework," tech. rep., RFC 6749, October 2012.

[56] S. Ince, D. Espes, J. Lallet, G. Gogniat, and R. Santoro, "Oauth 2.0-based authentication solution for fpga-enabled cloud computing," in *3rd International Workshop on Cloud, IoT and Fog Systems (and Security) - CIFS 2021 co-located with the 14th IEEE/ACM International Conference on Utility and Cloud Computing - UCC 2021*, (University of Leicester, UK), December 6–9 2021.

[57] Y. Hori, A. Satoh, H. Sakane, and K. Toda, "Bitstream encryption and authentication with aes-gcm in dynamically reconfigurable systems," in *2008 International Conference on Field Programmable Logic and Applications*, pp. 23–28, 2008.

[58] F. Devic, L. Torres, and B. Badrignans, "Secure protocol implementation for remote bitstream update preventing replay attacks on fpga," in *2010 International Conference on Field Programmable Logic and Applications*, pp. 179–182, 2010.

[59] A. Carelli, C. A. Cristofanini, A. Vallero, C. Basile, P. Prinetto, and S. Di Carlo, "Securing bitstream integrity, confidentiality and authenticity

in reconfigurable mobile heterogeneous systems," in *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pp. 1–6, 2018.

[60] U. Rührmair and et al., "Puf modeling attacks on simulated and silicon data," *IEEE Transactions on Information Forensics and Security*, vol. 8, pp. 1876–1891, Nov. 2013.

[61] J. Delvaux, "Machine-learning attacks on polypufs, ob-pufs, rpufs, lhs-pufs, and puf–fsms," *IEEE Transactions on Information Forensics and Security*, vol. 14, pp. 2043–2058, Aug. 2019.

[62] J. Delvaux, *Security Analysis of PUF-Based Key Generation and Entity Authentication*. PhD thesis, PhD Thesis, 2017.

[63] M. E. S. Elrabaa, M. Al-Asli, and M. Abu-Amara, "Secure computing enclaves using fpgas," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, pp. 593–604, March-April 2021.

[64] S. Zeitouni, J. Vliegen, T. Frassetto, D. Koch, A.-R. Sadeghi, and N. Mentens, "Trusted configuration in cloud fpgas," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 233–241, 2021.

[65] H. Xiao, K. Li, and M. Zhu, "Fpga-based scalable and highly concurrent convolutional neural network acceleration," in *2021 IEEE International Conference on Power Electronics, Computer Applications (ICPECA)*, pp. 367–370, 2021.

[66] T.-H. Tsai, Y.-C. Ho, and M.-H. Sheu, "Implementation of fpga-based accelerator for deep neural networks," in *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pp. 1–4, 2019.

[67] Y. Zhou and J. Jiang, "An fpga-based accelerator implementation for deep convolutional neural networks," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, pp. 829–832, 2015.

[68] Y. Zha and J. Li, "Virtualizing fpgas in the cloud," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 845–858, Association for Computing Machinery, 2020.

[69] M. K. Ahmed, J. Mandebi, S. K. Saha, and C. Bobda, "Multi-tenant cloud fpga: on security," 2022.

[70] S. Ince, D. Espes, G. Gogniat, R. Santoro, and J. Lallet, "Token-based authentication and access delegation for hw-accelerated telco cloud solution," in

2022 IEEE 11th International Conference on Cloud Networking (CloudNet), pp. 109–117, 2022.

[71] J. M. Mbongue, D. T. Kwadjo, A. Shuping, and C. Bobda, "Deploying multi-tenant fpgas within linux-based cloud infrastructure," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, dec 2021.

[72] S. Zeng, G. Dai, K. Zhong, H. Sun, G. Ge, K. Guo, Y. Wang, and H. Yang, "Enable efficient and flexible fpga virtualization for deep learning in the cloud," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, (New York, NY, USA), p. 317, Association for Computing Machinery, 2020.

[73] G. Dessouky, A.-R. Sadeghi, and S. Zeitouni, "Sok: Secure fpga multi-tenancy in the cloud: Challenges and opportunities," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 487–506, 2021.

[74] J. M. Mbongue, A. Shuping, P. Bhowmik, and C. Bobda, "Architecture support for fpga multi-tenancy in the cloud," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 125–132, 2020.

[75] O. Knodel, P. Lehmann, and R. G. Spallek, "Rc3e: reconfigurable accelerators in data centres and their provision by adapted service models," in *IEEE International Conference on Cloud Computing (CLOUD)*, pp. 19–26, 2016.

[76] J. Lallet, A. Enrici, and A. Saffar, "Fpga-based system for the acceleration of cloud microservices," in *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pp. 1–5, 2018.

[77] B. Janßen, F. Korkmaz, H. Derya, M. Hübner, M. L. Ferreira, and J. C. Ferreira, "Towards a type 0 hypervisor for dynamic reconfigurable systems," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–7, 2017.

[78] "vsphere." `https://www.vmware.com/products/vsphere.html`. Accessed on July 21, 2023.

[79] "Hyper-v." `https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/`. Accessed on July 21, 2023.

[80] "Kvm." `https://www.linux-kvm.org/`. Accessed on July 21, 2023.

[81] "Xen." `https://xenproject.org/`. Accessed on July 21, 2023.

[82] "Docker." `https://www.docker.com/`. Accessed on July 21, 2023.

[83] "Podman." `https://podman.io/`. Accessed on July 21, 2023.

[84] "Containerd." `https://containerd.io/`. Accessed on July 21, 2023.

[85] "Kubernetes." `https://kubernetes.io/`. Accessed on July 21, 2023.

[86] "Docker swarm." `https://docs.docker.com/engine/swarm/`. Accessed on July 21, 2023.

[87] "Apache mesos." `https://mesos.apache.org//`. Accessed on July 21, 2023.

[88] A. Vaishnav, K. D. Pham, and D. Koch, "A survey on fpga virtualization," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 131–1317, 2018.

[89] O. Knodel, P. R. Genßler, F. Erxleben, and R. G. Spallek, "Fpgas and the cloud – an endless tale of virtualization, elasticity and efficiency," 2019.

[90] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, "Fos: A modular fpga operating system for dynamic workloads," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, sep 2020.

[91] A. Brant and G. G. Lemieux, "Zuma: An open fpga overlay architecture," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 93–96, 2012.

[92] O. Knodel, P. Genssler, and R. Spallek, "Virtualizing reconfigurable hardware to provide scalability in cloud architectures," 09 2017.

[93] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. Reinhardt, A. Caulfield, E. Chung, and D. Burger, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th International Symposium on Computer Architecture, 2018*, ACM, June 2018.

[94] D. R. E. Gnad, V. Meyers, N. M. Dang, F. Schellenberg, A. Moradi, and M. B. Tahoori, "Stealthy logic misuse for power analysis attacks in multi-tenant fpgas," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1012–1015, 2021.

[95] J. Krautter, D. R. E. Gnad, and M. B. Tahoori, "Mitigating electrical-level attacks towards secure multi-tenant fpgas in the cloud," vol. 12, aug 2019.

[96] O. Glamočanin, A. Kostić, S. Kostić, and M. Stojilović, "Active wire fences for multitenant fpgas," in *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 13–20, 2023.

[97] J. Krautter, D. R. Gnad, F. Schellenberg, A. Moradi, and M. B. Tahoori, "Active fences against voltage-based side channels in multi-tenant fpgas," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2019.

[98] Y. Luo, C. Gongye, Y. Fei, and X. Xu, "Deepstrike: Remotely-guided fault injection attacks on dnn accelerator in cloud-fpga," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 295–300, 2021.

[99] G. Provelengios, D. Holcomb, and R. Tessier, "Power wasting circuits for cloud fpga attacks," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 231–235, 2020.

[100] T. La, K. Pham, J. Powell, and D. Koch, "Denial-of-service on fpga-based cloud infrastructures — attack and defense," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, p. 441–464, Jul. 2021.

[101] M. M. Alam, S. Tajik, F. Ganji, M. Tehranipoor, and D. Forte, "Ram-jam: Remote temperature and voltage fault attack on fpgas using memory collisions," in *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 48–55, 2019.

[102] C. Jin, V. Gohil, R. Karri, and J. Rajendran, "Security of cloud fpgas: A survey," *CoRR*, vol. abs/2005.04867, 2020.

[103] I. Giechaskiel, K. B. Rasmussen, and J. Szefer, "Measuring long wire leakage with ring oscillators in cloud fpgas," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 45–50, 2019.

[104] T. Sugawara, K. Sakiyama, S. Nashimoto, D. Suzuki, and T. Nagatsuka, "Oscillator without a combinatorial loop and its threat to fpga in data centre," *Electronics Letters*, vol. 55, no. 11, pp. 640–642, 2019.

[105] J. J. Rodríguez-Andina, M. D. Valdés-Peña, and M. J. Moure, "Advanced features and industrial applications of fpgas—a review," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 853–864, 2015.

[106] H. Nassar, H. AlZughbi, D. R. E. Gnad, L. Bauer, M. B. Tahoori, and J. Henkel, "Loopbreaker: Disabling interconnects to mitigate voltage-based attacks in multi-tenant fpgas," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2021.

[107] G. Provelengios, D. Holcomb, and R. Tessier, "Mitigating voltage attacks in multi-tenant fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 14, jul 2021.

[108] S. Donchez and X. Wang, "Memory isolation for multi-tenant data integrity in cloud mpsoc fpgas," in *2022 IEEE 13th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pp. 0515–0521, 2022.

[109] E. Karabulut, A. Awad, and A. Aysu, "Ss-axi: Secure and safe access control mechanism for multi-tenant cloud fpgas," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2023.

[110] I. Giechaskiel, S. Tian, and J. Szefer, "Cross-vm covert- and side-channel attacks in cloud fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, dec 2022.

[111] H. Oh, K. Nam, S. Jeon, Y. Cho, and Y. Paek, "Meetgo: A trusted execution environment for remote applications on fpga," *IEEE Access*, vol. 9, pp. 51313–51324, 2021.

[112] Z. Zhu, A. X. Liu, F. Zhang, and F. Chen, "Fpga resource pooling in cloud computing," *IEEE Transactions on Cloud Computing*, vol. 9, pp. 610–626, April-June 2021.

[113] E. Karabulut, A. Awad, and A. Aysu, "Ss-axi: Secure and safe access control mechanism for multi-tenant cloud fpgas," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2023.

[114] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci, "A hypervisor for shared-memory fpga platforms," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 827–844, Association for Computing Machinery, 2020.

[115] Y. Zha and J. Li, "Virtualizing fpgas in the cloud," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 845–858, 2020.

[116] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with amorphos," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 107–127, USENIX Association, October 2018.

[117] D. Korolija, T. Roscoe, and G. Alonso, "Do os abstractions make sense on fpgas?," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 991–1010, USENIX Association, November 2020.

[118] "Json web token." `https://jwt.io/`.

[119] M. Jones, J. Bradley, and N. Sakimura, "Json web token (jwt)," RFC 7519, Internet Engineering Task Force (IETF), 2015.

[120] "O-ran documentation." `https://docs.o-ran-sc.org/en/latest/index.html`.

[121] F. Schellenberg, D. R. E. Gnad, A. Moradi, and M. B. Tahoori, "An inside job: Remote power analysis attacks on fpgas," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1111–1116, 2018.

[122] M. Zhao and G. E. Suh, "Fpga-based remote power side-channel attacks," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 229–244, 2018.

[123] Y. Luo, C. Gongye, S. Ren, Y. Fei, and X. Xu, "Stealthy-shutdown: Practical remote power attacks in multi-tenant fpgas," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, (Hartford, CT, USA), pp. 545–552, 2020.

[124] S. Moini, S. Tian, D. Holcomb, J. Szefer, and R. Tessier, "Remote power side-channel attacks on bnn accelerators in fpgas," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, (Grenoble, France), pp. 1639–1644, 2021.

[125] "Op-tee, a tee-based operating system." `https://www.op-tee.org/`.

[126] M. Grinberg, "Flask: A lightweight web application framework," 2018. `https://palletsprojects.com/p/flask/`.

[127] C. Herder, M. Yu, F. Koushanfar, and S. Devadas, "Physical unclonable functions and applications: A tutorial," *Proceedings of the IEEE*, vol. 102, pp. 1126–1141, Aug. 2014.

[128] O. Mutlu, "The rowhammer problem and other issues we may face as memory becomes denser," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 1116–1121, 2017.

[129] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, "Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms," *arXiv preprint arXiv:1912.11523*, 2019.

[130] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 638–651, 2020.

[131] O. Petura, U. Mureddu, N. Bochard, V. Fischer, and L. Bossuet, "A survey of ais-20/31 compliant trng cores suitable for fpga devices," in *2016 International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 1–10, 2016.

[132] E. Barker and J. Kelsey, "Nist released special publication (sp) 800-90a revision 1: Recommendation for random number generation using deterministic random bit generators," *National Institute of Standards and Technology*, June 2015.

[133] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management, part 1: General," *NIST Special Publication*, pp. 51–54, 2016.

[134] O.-R. W. G. 11, "Security requirements specifications v7.0," tech. rep., O-RAN, 2023. Version 7.0.

[135] O.-R. W. G. 11, "O-ran security threat modeling and remediation analysis," tech. rep., O-RAN, Alfter, Germany, 2023. Version 6.0.

[136] O.-R. W. G. 11, "Study on security for o-cloud," tech. rep., O-RAN, 2023. Version 4.0.

[137] 3rd Generation Partnership Project, "3gpp tr 33.848 v0.14.0. technical specification group services and system aspects; security aspects; study on security impacts of virtualisation (release 18)," Technical Report 33.848, Technical Specification Group Services and System Aspects, 2023. Version 0.14.0.

[138] N. Giri and N. N. Anandakumar, "Design and analysis of hardware trojan threats in reconfigurable hardware," in *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, (Vellore, India), pp. 1–5, 2020.

[139] W. Danesh, J. Banago, and M. Rahman, "Turning the table: Using bitstream reverse engineering to detect fpga trojans," *Journal of Hardware Systems and Security*, vol. 5, pp. 237–246, 2021.

[140] R. Mukherjee and R. S. Chakraborty, "Novel hardware trojan attack on activation parameters of fpga-based dnn accelerators," *IEEE Embedded Systems Letters*, vol. 14, pp. 131–134, September 2022.

[141] B. Hanindhito, N. Ahmadi, H. Hogantara, A. Arrahmah, and T. Adiono, "Fpga implementation of modified serial montgomery modular multiplication for 2048-bit rsa cryptosystems," May 2015.

[142] M. M. Sadeghi, S. Timarchi, and M. Fazlali, "High-performance memory allocation on fpga with reduced internal fragmentation," *IEEE Access*, vol. 11, pp. 66672–66681, 2023.

[143] T. Liang, J. Zhao, L. Feng, S. Sinha, and W. Zhang, "Hi-dmm: High-performance dynamic memory management in high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2555–2566, 2018.

[144] Z. Xue and D. B. Thomas, "Sysalloc: A hardware manager for dynamic memory allocation in heterogeneous systems," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–7, 2015.

[145] R. Ammendola, A. Biagioni, O. Frezza, F. L. Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "Virtual-to-physical address translation for an fpga-based interconnect with host and gpu remote dma capabilities," in *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 58–65, 2013.

[146] M. Kaiiali, S. Sezer, and A. Khalid, "Cloud computing in the quantum era," in *2019 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–4, 2019.

[147] M. Mosca, "Cybersecurity in an era with quantum computers: Will we be ready?," *IEEE Security & Privacy*, vol. 16, no. 5, pp. 38–41, 2018.

[148] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, 2017.

[149] "Preparing for post-quantum cryptography." `https://www.ncsc.gov.uk/whitepaper/next-steps-preparing-for-post-quantum-cryptography`.

PUBLICATIONS

Apart from the primary content presented in this thesis, other works have also been published. They are listed below in a chronological order with a brief summary.

1. Semih Ince, David Espes, Guy Gogniat, Julien Lallet, and Renaud Santoro. 2022. OAuth 2.0-based authentication solution for FPGA-enabled cloud computing. In Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC '21). Association for Computing Machinery, New York, NY, USA, Article 13, 1–6.
   https://doi.org/10.1145/3492323.3495635

   Summary: This work provided a token-based FPGA access framework by adapting OAuth 2 to FPGA clouds. At the time of writing, no other token-based FPGA access scheme exists. This work laid the foundation of TokSek by providing an architecture for FPGA cloud. The described framework leverages a trusted authority to act as a security anchor. This work gives an emphasis on FPGA-user authentication as it is a crucial element to establish a secure acceleration environment.

2. Semih Ince, David Espes, Guy Gogniat, Renaud Santoro and Julien Lallet, Token-based authentication and access delegation for HW-accelerated telco cloud solution, 2022 IEEE 11th International Conference on Cloud Networking (CloudNet), Paris, France, 2022, pp. 109-117,
   DOI: 10.1109/CloudNet55617.2022.9978865.

   Summary: This work extends the TokSek framework capabilities by allowing a user to share its allocated device with another user. This mechanism is particularly applicable in telecommunication context where one network provider wants to give network access to another provider. Additionally, experimental results of TokSek are exposed and a low latency solution is showcased.

3. Semih Ince, David Espes, Guy Gogniat, Renaud Santoro and Julien Lallet, "Authentication and Confidentiality in FPGA-based Clouds" in "Security of FPGA-Accelerated Cloud Computing Environments," J. Szefer and R. Tessier, Eds. Springer, 1st ed., 2024, pp. 1-27, DOI: 10.1007/978-3-030-14540-0

Summary: In this book chapter, an extensive state of the art is provided to highlight the importance of authentication in FPGA clouds. In fact, authentication techniques are described and compared against the deployments in the literature. Direct authentication and authentication with a third-party is explained as it is a popular method in FPGA cloud literature. Various drawbacks are identified in and some mitigation from the literature are proposed as a solution. Then, the Toksek framework is proposed to address some of the identified challenges.

4. Semih Ince, David Espes, Guy Gogniat, Renaud Santoro and Julien Lallet, "Linkguard", patent registration number: 20235874, https://patenttitietopalvelu.prh.fi/en/

   Summary: Linkguard is a patented hardware-based security solution that creates confidential channels to third-party cloud-based FPGAs without relying on the cloud provider. The hardware-based protocol ensures isolation from other stakeholders and protects user data in the FPGA.

List of Figures

List of Tables

COLLEGE MATHSTIC
DOCTORAL BRETAGNE
BRETAGNE OCEANE

Université de Bretagne Occidentale

Titre : Framework de sécurité avec token pour de l'accélération cloud FPGA multi-utilisateurs sécurisée basé

Mots clés : FPGA, sécurité, cloud,

Résumé: La demande croissante en puissance de calcul provient de la convergence d'applications intensives en données telles que l'apprentissage automatique, l'intelligence artificielle et l'analyse de big data, ainsi que de l'essor des technologies comme la 5G, l'Internet des objets (IoT) et les systèmes autonomes. L'informatique en nuage offre une solution évolutive et à la demande pour ces besoins en calcul haute performance, permettant aux organisations de tirer parti d'environnements virtualisés sans avoir à investir massivement dans des infrastructures matérielles dédiées. Parmi les dispositifs de calcul, les Field-Programmable Gate Arrays (FPGAs) se distinguent par leur reconfigurabilité et leur capacité à accélérer efficacement des charges de travail spécifiques, offrant des avantages par rapport aux processeurs (CPU) et aux unités de traitement graphique (GPU).

Dans les environnements de cloud FPGA multi-utilisateurs, il est crucial de garantir la confidentialité des données, l'isolement des utilisateurs et un accès sécurisé. Cette thèse présente **TokSek**, un cadre de sécurité pour les clouds FPGA multi-utilisateur, visant à protéger les données des utilisateurs tout en maintenant un faible impact sur les performances. **TokSek** étend le cadre OAuth 2 pour un accès sécurisé aux FPGAs via une authentification par jeton et applique le contrôle d'accès à travers un module de sécurité matériel. De plus, le protocole breveté **Linkguard** établit un canal de communication confidentiel basé sur une approche de zéro confiance entre les utilisateurs et les FPGAs dans des environnements de cloud non sécurisés.

Title: TokSek: Token-based multi-tenant cloud FPGA security framework for secure acceleration

Keywords : FPGA, security, cloud, token

Abstract: The growing demand for computing power stems from the convergence of data-intensive applications like machine learning, artificial intelligence, and big data analytics, alongside the rise of technologies such as 5G, IoT, and autonomous systems. Cloud computing provides a scalable, on-demand solution for these high-performance computing needs, allowing organizations to leverage virtualized environments without heavy investments in dedicated hardware. Among computing devices, Field-Programmable Gate Arrays (FPGAs) stand out due to their reconfigurability and ability to accelerate specific workloads efficiently, offering advantages over CPUs and GPUs

In multi-tenant FPGA cloud environments, ensuring data confidentiality, user isolation, and secure access is critical. This thesis introduces **TokSek**, a security framework for multi-tenant FPGA clouds aimed at protecting user data while maintaining low performance overhead. **TokSek** extends the OAuth 2 framework for secure FPGA access through token-based authentication and enforces access control via a hardware security module. Additionally, the patented **Linkguard** protocol establishes a zero-trust confidential communication channel between users and FPGAs in untrusted cloud environments.