Salus: A Practical Trusted Execution Environment for CPU-FPGA Heterogeneous Cloud Platforms

Yu Zou
zouyu.zou@alibaba-inc.com
Alibaba Group

Yiran Li
yiranli.lyr@alibaba-inc.com
Alibaba Group

Sheng Wang*
sh.wang@alibaba-inc.com
Alibaba Group

Le Su
le.su@alibaba-inc.com
Alibaba Group

Zhen Gu
guzhen.gz@alibaba-inc.com
DAMO Academy, Alibaba Group
Hupan Lab

Yanheng Lu
yanheng.lyh@alibaba-inc.com
DAMO Academy, Alibaba Group
Hupan Lab

Yijin Guan
yijin.gyj@alibaba-inc.com
DAMO Academy, Alibaba Group
Hupan Lab

Dimin Niu
dimin.niu@alibaba-inc.com
DAMO Academy, Alibaba Group
Hupan Lab

Mingyu Gao
gaomy@tsinghua.edu.cn
Tsinghua University
Shanghai AI Laboratory
Shanghai Qi Zhi Institute

Yuan Xie
y.xie@alibaba-inc.com
DAMO Academy, Alibaba Group
Hupan Lab

Feifei Li
lifeifei@alibaba-inc.com
Alibaba Group

Abstract

CPU-FPGA heterogeneous architectures have become increasingly popular in cloud environments for accelerating compute-intensive tasks. Ensuring the protection of sensitive data processed by these architectures requires the presence of a trusted execution environment (TEE). This work highlights the requirements for designing an FPGA TEE, the challenges faced in deploying existing solutions on commercial-off-the-shelf (COTS) cloud FPGA services, and the limitations of previous works that primarily focus on standalone FPGA TEEs. In response to these challenges, *Salus* introduces an innovative approach by leveraging an enclave running on the host with a TEE-enabled CPU. This approach aims to protect and attest the bitstream loaded on the FPGA side. By repurposing COTS FPGA bitstream utilities in a novel manner and adopting a proposed security-enhanced FPGA IP, *Salus* presents a practical design for an FPGA TEE, with minor efforts required.

*Sheng Wang is the corresponding author of this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0391-1/24/04
https://doi.org/10.1145/3622781.3674169

ACM Reference Format:
Yu Zou, Yiran Li, Sheng Wang, Le Su, Zhen Gu, Yanheng Lu, Yijin Guan, Dimin Niu, Mingyu Gao, Yuan Xie, and Feifei Li. 2024. *Salus*: A Practical Trusted Execution Environment for CPU-FPGA Heterogeneous Cloud Platforms. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3622781.3674169

1 Introduction

In the cloud, a high volume of user data flows through shared infrastructures maintained by cloud service providers (CSPs). It is crucial to prioritize data privacy when constructing these cloud infrastructures. Trusted execution environment (TEE) offers a secure solution to prevent data breaches, by providing a hardware-isolated environment for executing programs, known as enclaves.

In addition to data privacy, computing efficiency is a significant focus in cloud computing. Prominent CSPs like AWS [1], Azure [8], and Alibaba [10] offer FPGA-as-a-Service (FaaS), enabling users to efficiently offload computing tasks to hardware accelerators. FaaS partitions an FPGA device into a shell and custom logic (CL). Users deploy their own accelerator logic onto the CL, while the CSP-maintained shell functions as a privileged OS, responsible for CL deployment, I/O monitoring, and resource management. Beyond inheriting vulnerabilities from CPU standalone platforms, FaaS introduces new vulnerabilities on the FPGA side caused by the CSP-maintained privileged shell. Therefore it is crucial to establish a TEE on the FPGA side.

The CL on the FPGA side encounters a similar attack surface as a program executed on the host side. Firstly, the privileged shell can potentially compromise the integrity of the CL bitstream during the loading process. Secondly, the shell has the ability to monitor CL's I/O, compromising data confidentiality and integrity. To address these concerns, an FPGA TEE must meet two requirements: 1) the provision of a root-of-trust (RoT) and an attestation mechanism enabling the user on the host side to verify the integrity of the loaded CL bitstream, and 2) an I/O protection mechanism ensuring data confidentiality and integrity. While integrating I/O encryption within the CL can guarantee I/O protection, fulfilling the RoT and attestation requirement poses challenges.

Existing COTS FPGAs do not posses an on-board RoT which is exclusively accessible to the attestation logic. Without a verifiable RoT to identify the message issuer, the CL attestation is susceptible to confidentiality attacks and integrity attacks issued from the privileged shell. Previous efforts to design FPGA TEEs either rely on additional secure hardware as a RoT [22, 31, 42] or incorporate a hardcoded secret, *e.g.* a PUF-generated challenge-response database, bundled to a specific device as a RoT [40]. Consequently, they are neither cost-efficient (cannot be applied to COTS FPGA devices) nor compatible with current cloud FPGA usage (where a CL should be general enough to be deployed on any cloud FPGA device). As a result, these approaches cannot be directly implemented in current legacy FaaS.

We present *Salus*, an innovative FPGA TEE design that prioritizes *practicality and compatibility* for easy implementation on COTS cloud FPGA devices. Our approach is based on two key observations:

Observation 1: CPU TEEs are readily available in current cloud infrastructure. Existing cloud platforms [5, 7] are already equipped with CPU TEEs. Therefore, it is more practical to utilize these pre-existing CPU TEEs rather than relying on additional secure RoT hardware.

Observation 2: FPGA dynamic partial reconfiguration overwrites all logic cells within the dynamic area, leaving no unchanged logic cells. The partial bitstream generated on existing FPGAs cover the configuration for every cells within the dynamic area even though not used by the custom logic [2].

Relying on the above two observations, we target a heterogeneous secure system consisting of a CPU TEE and an FPGA TEE. Rather than relying on an extra RoT or hard-coded secrets, *Salus* utilizes a CPU enclave on the host side in an inventive manner. Specifically, an IP developer develops a CL containing a general attestation logic with a reserved storage for the RoT during the development. During the deployment, a verifiable RoT is dynamically generated and injected inside the CL within a CPU enclave. As the RoT is securely generated within the enclave, the enclave could cooperate with the attestation logic on the CL to guarantee that a CL containing the correct RoT is loaded. During the CL loading, we keep the CL bitstream confidential to the shell, such that a malicious attacker could not reverse engineer a plaintext CL bitstream, replace the CL while leaving the RoT unchanged to fake a valid CL attestation. Since the second observation ensures that the CL is always programmed as an entirety, the integrity of the RoT further indicates the integrity of the entire CL.

To fulfill the goal, *Salus* faces three technical challenges:

Challenge 1: Fast and secure dynamic injection of a RoT inside the bitstream. The injected RoT must be confidential to prevent the privileged shell's snooping attacks. A naive approach would require the IP developer open-sources the accelerator code to the IP user and the latter dynamically hard-codes a RoT by changing the source code, and generates a new bitstream inside the host enclave during the deployment phase. However, this approach is time-consuming due to placement and routing, resulting in unacceptable deployment delays. In addition, requiring the IP developer open-source the IP to the IP user is not compatible with the cloud usage, where the IP developer and the IP user might belong to different entities.

Solution 1: Innovative usage of bitstream manipulation and encryption. *Salus* utilizes existing bitstream manipulation and encryption techniques in a novel way. During the booting process, the host enclave uses bitstream manipulation to inject a random secret into the CL bitstream. The bitstream manipulation operates a bitstream directly on the bitstream level. The bitstream is then encrypted within the enclave and loaded by the shell. The FPGA's internal decryption mechanism ensures that the shell cannot steal the embedded secret during bitstream loading. Utilizing the bitstream manipulation and encryption, the RoT injection does not need to go through the time-consuming placement and routing, and the RoT can be dynamically generated and embedded during the deployment phase. By further disabling FPGA's internal readback capability, the embedded secret is fully isolated from the shell during runtime. *Salus* is able to utilize these techniques since they are either already available in modern FPGAs (*e.g.*, bitstream encryption and manipulation) or feasible with FPGA vendors' involvement (*e.g.*, releasing a new bitstream loading RTL IP with readback disabled), enabling their implementation on COTS FPGA devices. In addition, eliminating the need of source code open-sourcing, *Salus* is compatible with the cloud usage.

Challenge 2: Attestation of custom logic. The host enclave cooperates with the attestation logic integrated on the CL to authenticate the embedded RoT to ensure that the RoT and attestation logic have not been tampered with, guaranteeing the secure loading of the bundled CL bitstream. As all the transactions between the host and the attestation logic are carried by a malicious shell, an light-weight attestation mechanism resistant to typical bus attacks, *e.g.*, confidentiality and integrity attacks, is desired. ShEF [42] proposes a CL attestation framework analogous to CPU TEE's remote attestation. The public-key encryption (PKE) based remote attestation requires a public key infrastructure, which highly complicates the system design. Additionally, ShEF requires each CL developer to work as a certificate authority (CA) to verify the bitstream, resulting in the developer's involvement in the cloud deployment phase. Therefore, a simple and light-weight CL attestation mechanism is desired.

Solution 2: Light-weight custom logic attestation. *Salus* utilizes the host enclave to attest the CL. As the host enclave can be remotely attested and trusted, it is only required to verify the CL from the enclave. Utilizing the trusted host enclave, *Salus* proposes a new CL attestation mechanism analogous to the local attestation used in CPU TEEs. Unlike the remote attestation, the CPU TEE's local attestation utilizes symmetric key based verification, which is highly compatible with the proposed RoT injection. By injecting the RoT within the host enclave, the injected RoT is secure and can be utilized directly as a symmetric key, thereby simplifying the attestation process and eliminating the need for an additional CA and another PKE round, as in the case of ShEF. *Salus* utilizes a light-weight message authentication code to securely attest the CL over an unsecure channel.

Challenge 3: Authentication of multiple heterogeneous components in the system. A secure heterogeneous system consists of multiple components, including both host enclaves and CL. Private data must flow through the system only when all the components are trusted. Therefore, when a cloud customer receives an attestation report before uploading sensitive data, this report must assure that all components have undergone attestation. Prior attestation protocols for heterogeneous systems, *e.g.*, the multi-stage attestation used in SGX-FPGA [40], only generate an attestation report covering the security of a subset of components. Consequently, upon receipt of the report, the cloud customer cannot immediately proceed with uploading sensitive data as the verification process for all components is yet to be completed. To the best of our knowledge, no secure attestation protocol has been proposed that provides a report containing a security proof for all heterogeneous components.

Solution 3: Cascaded attestation. To ensure secure authentication of all components in a heterogeneous system, *Salus* proposes a *cascaded attestation* protocol. This protocol chains and cascades the authentication process from the user enclave on the host to the CL on the FPGA. This allows the entire system to be remotely attested by the cloud customer in a single round-trip.

In summary, we make following contributions:

- We introduce *Salus* as the first cost-efficient TEE for a CPU-FPGA heterogeneous system, which could be directly applied to existing FaaS platforms.

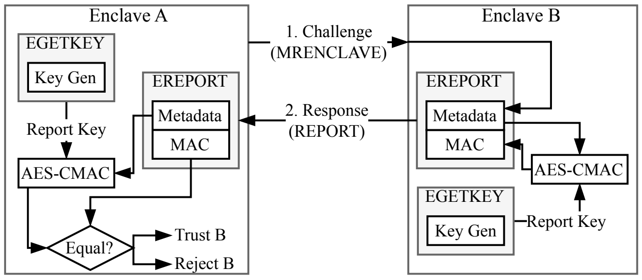


Figure 1. Local attestation protocol of Intel SGX.

- We explore the repurposed usage of existing FPGA bitstream utilities to securely embed a RoT into a CL bitstream guaranteeing confidentiality and integrity of the CL loaded by a potentially malicious shell.
- We propose a light-weight CL attestation protocol allowing for authenticating a loaded CL over a channel vulnerable to shell's attacks, without the need of complex PKE.
- We design a novel cascaded attestation protocol allowing for generating an attestation report containing security proofs for all heterogeneous infrastructure components for the cloud customer to verify.
- We implement a prototype on a CPU-FPGA platform and evaluate using five real-world benchmarks. The secure booting time for an FPGA TEE is measured to be 18.1 seconds, which is reasonable compared to the overall booting time of a cloud VM instance. Additionally, *Salus* demonstrates significant speedup (up to 15.6 times) compared to running within a CPU enclave in real-world applications.

2 Background

2.1 Trusted Execution Environment

A TEE like Intel SGX in general consists of three functionalities: a root-of-trust (RoT), execution isolation, and remote attestation. The TEE uses a manufacturer-injected key as a RoT. An enclave program is loaded into a hardware-enforced isolated memory region. The TEE generates a proof for a verifier to examine the authenticity of an enclave, which is called remote attestation (RA). A successful RA gives the verifier an attestation report, *e.g.*, a Data Center Attestation Primitive (DCAP) quote for Intel SGX, indicating that the enclave runs on a fully patched TEE platform and therefore the verifier can pass secrets to the enclave.

In addition to RA, TEE implementations also provide a local attestation mechanism proving that two enclaves run on the same platform. As shown in Figure 1, two hardware instructions, `EGETKEY` and `EREPORT`, are used in the local attestation [19]. `EGETKEY` guarantees that only a trusted enclave could get a symmetric report key, and `EREPORT` generates a report signed by the report key. Following a challenge-response roundtrip, the verifier enclave compares the locally generated attestation report with that of the prover enclave. A successful verification guarantees that the prover enclave has access to the report key and consequently runs on the same platform as the verifier enclave.

Typical CPU TEEs, *e.g.*, Intel SGX, only provide a static attestation, that is the attestation only verifies the static environment of an enclave, while not providing defense to attacks aiming to modify the program runtime behavior. Similarly, *Salus* only focuses on protecting integrity of the CL during bitstream loading, ignoring runtime attacks, *e.g.*, runtime bitstream replacement. Runtime attestation like those proposed for CPU TEEs [26, 30, 38] will be studied later.

2.2 FPGA-as-a-Service

FaaS utilizes dynamic partial reconfiguration to split a device into a static partition, running a CSP-maintained shell, and a reconfiguration partition (RP), loaded by various custom logics (CLs). The shell is responsible for programming an RP with a CL bitstream and abstracting away complex control logic of external devices, such as DRAM and PCIe, acting as an OS. The shell uses a special on-board IP to interface with the FPGA configuration memory, referred as Internal Configuration Access Port (ICAP) for Xilinx FPGAs. During development, IP vendors uses CSP-licensed hardware development kit (HDK) and software development kit (SDK) to design accelerators and generates partial bitstreams and host program binaries. During a cloud instance creation, the CSP deploys a privileged OS and other programs on the host, and loads the shell into the FPGA. Following the creation, the instance customer deploys the data processing program on the host, and transfers the CL bitstream to the shell, later loaded into the FPGA. After the deployment, the customer uploads data to the cloud for heterogeneous processing.

2.3 Bitstream Encryption And Manipulation

Bitstream encryption. Existing FPGA devices support bitstream encryption and an AES decryption key can be stored internally in either a battery-backed on-chip RAM (BBRAM) or an eFUSE. During development, the developer generates a key and encrypts the bitstream. During deployment, the IP user fuses the key to the FPGA via a JTAG interface and loads the encrypted bitstream. The bitstream decryption occurs internally within the FPGA fabric and remains inaccessible to FPGA users, guaranteeing the bitstream's confidentiality. Unlike the FPGA logic programmed by any FPGA user, the FPGA fabric is released only by the manufacturer thus the internal decryption is considered trusted according to the threat model in Section 3. Note that the bitstream encryption is compatible with partial reconfiguration allowing a shell to load an encrypted bitstream. However, this encryption feature is originally designed for on-premise usage, where the IP developer and IP user belong to the same entity, and not suitable for a secure cloud environment Firstly, in a shared resource like a cloud FPGA, exclusive loading of a key impedes resource multiplexing. Secondly, the IP user and IP developer may be different entities, making the traditional bitstream encryption usage tightly couple the IP deployment phase (scheduled by the user) and the development phase (scheduled by the developer) to transfer an encryption key. Lastly, leaking an encryption key to a compromised IP user would negatively affect the benefit of the IP developer.

Bitstream manipulation. A bitstream is a sequence of initial values for configuration memory cells, such as registers and BRAMs. Bitstream manipulation takes a readily available FPGA bitstream and the hierarchical location of a specific cell in the generated netlist as inputs, and updates with a user-defined initialization value without the need to modify the RTL code. The bitstream manipulation capability is provided in both academic tools *e.g.*, byteman [28] and industrial products, *e.g.*, RapidWright [27].

3 Threat Model And Motivation

3.1 Threat Model

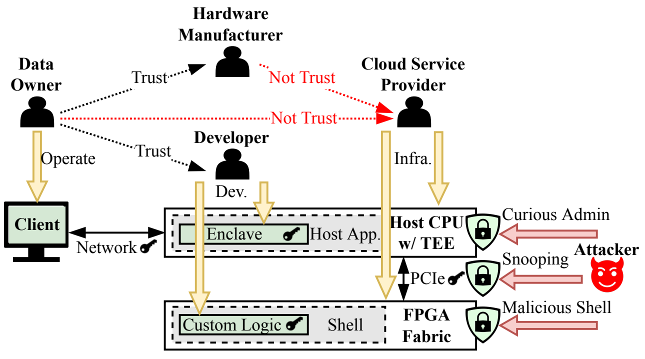


Figure 2. Threat model of secure FaaS.

As shown in Figure 2, *Salus* targets a cloud CPU-FPGA heterogeneous platform consisting of a host and a shell-managed FPGA. The target cloud usage considers four parties: data owner (also cloud user), developer, CSP, and hardware manufacturer. For simplicity, we do not distinguish between the host side and the FPGA side in this work. The goal of a secure FaaS is to provide a secure cloud platform such that a developer's program can be faithfully deployed on the platform and the data owner can attest the platform's state, upload sensitive data, and start executing the program.

The CSP hosts a host system enhanced by the CPU TEE and an FPGA enhanced by our proposed FPGA TEE. More details of the FPGA TEE will be provided in Section 4. The goal of this heterogeneous architecture is to prevent attacks issued from privileged adversaries, *e.g.*, CSP administrators. Typical attacks protected by *Salus* include:

1. Integrity attacks on CL during booting. A shell controlled by an adversary loads a malicious CL to steal the data owner's sensitive data.
2. Confidentiality and integrity attacks on CL during runtime. An adversary tampers with the device memory to steal user data or change control flow.
3. Bus attacks on host-CL PCIe transactions. Both attestation transactions over PCIe and runtime data transactions over PCIe suffer from the attacks.
4. Privileged attacks on the host trying to steal data or affect control flow of programs running on the host.

In this work, we assume all programs running inside the CPU TEE are tamper resistant to privileged attacks. We also delegate the task of data encryption and decryption to the developer who should implement corresponding modules in the CL to protect against device memory attacks. There are many research efforts targeting to provide efficient and flexible memory integrity and confidentiality protection [33, 34, 42, 45, 46]. As such, *Salus* only focuses on the secure loading of a CL and how to establish an encrypted channel between an enclave program running inside the host TEE and a CL running inside the FPGA TEE for trusted attestation and data transaction.

As mentioned in Section 2.2, the shell uses ICAP to internally interface with the FPGA configuration memory. This results in an attack surface that the shell could snoop a loaded CL by internally scanning the configuration memory. Consequently, any secrets hardcoded inside an RTL source code can be read back by the shell in plaintext. *Salus* assumes the ICAP readback capability disabled such that the shell could only load but not scan the CL. In Section 5.1.2, we'll argue that this is a feasible assumption with the FPGA vendors' involvement. Existing works also have similar implicit assumptions [22, 31, 40, 42].

Similar to the CPU TEE, we assume the hardware manufacturer is trusted by the data owner and we assume there is no backdoor logic injected during the device manufacturing process. Moreover, as for Intel SGX, we assume the manufacturer is trusted such that it could faithfully work as a verification authority. Secrets, *e.g.*, symmetric device keys used in bitstream encryption, are assumed securely stored in a trusted key management service by the manufacturer.

The developer is responsible for generating a host enclave program binary and a CL partial bitstream. We assume the program and the bitstream are developed under a secure environment and the implementation itself is trusted.

Throughout the work, side-channel attacks on either the CPU or the FPGA are not considered and *Salus* is compatible with other advanced mitigation measurements [21, 32]. Denial-of-Service (DoS) attacks are not considered, as the CSP is assumed to have a strong incentive to prevent DoS attacks to avoid business losses.

Table 1. Comparison with Existing FPGA TEE Works

Work	**TEE Type**	**No Extra Hardware**	**Independent Dev. & Dep.**[1]
SGX-FPGA [40]	HE[2]	✓	✗
ShEF [42]	SA[3]	✗	✓
MeetGo [31]	SA[3]	✗	✓
Ambassy [22]	SA[3]	✗	✓
Salus	HE[2]	✓	✓

[1] Independent IP development phase and deployment phase.
[2] HE: Heterogeneous CPU-FPGA TEE.
[3] SA: Standalone FPGA TEE.

3.2 Motivation

To build a RoT on the FPGA side to be used to attest the loaded CL bitstream, previous works, focusing to build a standalone FPGA TEE independent of the host, have typically required additional hardware to assist in the attestation process as shown in Table 1. ShEF [42] and Ambassy [22] rely an ARM processor and assume that a unique private key injected during the manufacturing process is hardcoded into each BootROM. MeetGo [31] proposes a new FPGA programming unit exposing a pre-injected private key to the CL. Therefore, these works cannot be directly applied to COTS FPGA devices. Similar to *Salus*, SGX-FPGA [40] also targets a CPU-FPGA heterogeneous system instead of standalone FPGA TEEs and it uses a physically unclonable function (PUF) challenge-response pair (CRP) database to use as a RoT. However, since the PUF is unique per device, the developer needs to operate on the FPGA board that the user intends to use to pre-generate a CRP database specific to the cloud instance. This contradicts typical cloud usage where IP development is independent of instance deployment.

Salus, on the other hand, takes a different approach by utilizing a host enclave to assist in CL booting and attestation. It innovatively repurposes bitstream manipulation and encryption to dynamically inject unique RoT into the CL per deployment eliminating the need for extra hardware. Moreover, the RoT injection approach decouples the development phase and the deployment phase, as the developer only needs to integrate a general CL attestation logic.

4 System Design

In this section, we present the system design. We provide an overview in Section 4.1. Then, we delve into RoT injection, CL attestation, and cascaded attestation, covered in Section 4.2, Section 4.3, and Section 4.4, respectively. We further explain the interface for the user application in Section 4.5. Section 4.6 gives security analysis. It is important to note that we use Xilinx FPGA and Intel SGX as examples and adopt their terminologies, but *Salus* is not device-bound and compatible with other FPGA and TEE implementations.

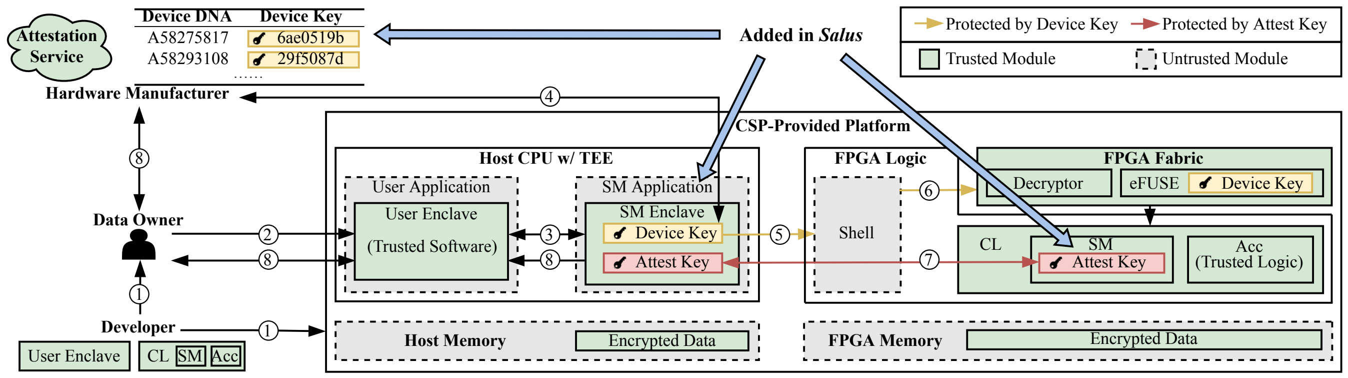


Figure 3. Secure RoT injection and CL booting flow of *Salus*.

4.1 System Overview

Salus utilizes bitstream manipulation and encryption to inject a RoT to the CL bitstream. As discussed in Section 2.3, the original bitstream encryption is not suitable for cloud usage. *Salus* repurposes the bitstream encryption scheme by involving a separate portable enclave application, in addition to the user enclave, responsible for bitstream encryption, which is developed by a trusted third-party. This third-party injects a secret key into each cloud FPGA device and maintains a key distribution service. The third-party securely issues the corresponding key for the FPGA device. While any trusted third-party could fulfill this role, in this work, *Salus* assigns this responsibility to the hardware manufacturer for simplicity. Additional CL secure booting related functionalities, like bitstream manipulation and attestation, are also offloaded to the enclave. Traditional CPU TEEs also assign similar responsibilities to the hardware manufacture, *e.g.*, Intel Attestation Service [4]. Existing FPGA vendors already offer similar key distribution services [9], thus providing the desired service is not burdensome for the manufacturer.

In general, in addition to the user enclave application and user accelerator, *Salus* adds extra three components: secure manager (SM) enclave application running on the host side alongside the user enclave, secure manager (SM) logic running within the CL alongside the accelerator logic, and a key distribution service maintained by the manufacturer as shown in Figure 3. The SM logic and accelerator are integrated during development, generating a single CL bitstream containing both logics. The SM application and logic are responsible for manipulating, encrypting, and attesting CL. They are expected to be released by the manufacturer as a software development kit (SDK) and a hardware development kit (HDK), respectively. Design details of the SM application and logic will be discussed in Section 5. Both SM application and SM logic solely utilize well-known cryptographic functionalities like AES encryption, SHA, and HMAC. They do not contain hardcoded secrets, ensuring a compact and easily inspectable codebase for secure implementation. It is easier for these HDK/SDK to be open-sourced and the developers verify before integrating into their own designs. Both SM enclave and logic are versatile, requiring development only once and suitable for all platforms.

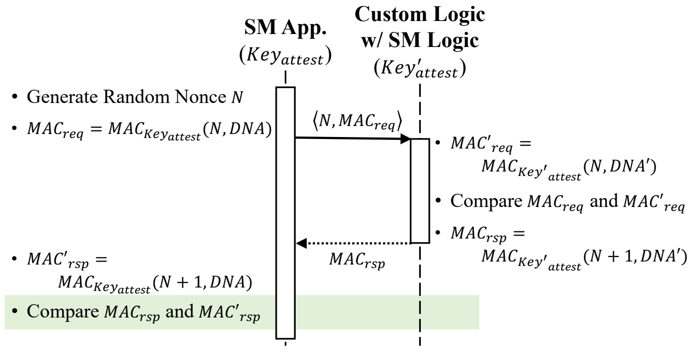
4.2 Dynamic RoT Injection And Secure CL Booting

Salus innovatively relies on bitstream manipulation and encryption to inject a RoT to the CL bitstream. However, the RoT needs to be injected to the CL bitstream securely such that only the SM enclave and the correct CL has access. Specifically, a CL booting flow with a secure RoT injection needs to guarantee the following: 1) A correct and user-expected bitstream is operated; 2) The encryption key associated to the FPGA device is not leaked to attackers; 3) The bitstream is securely manipulated and encrypted; and 4) The plaintext bitstream containing the injected RoT is not leaked.

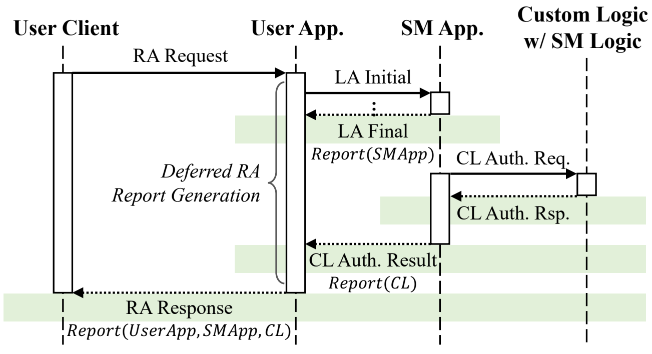
To fulfill the goal, we design a secure CL booting flow to guarantee that before the CL is deployed on the FPGA, a dynamic RoT is securely embedded into the bitstream. The RoT is not leaked to any privileged component such as OS and shell. In general, in addition to the normal CL booting flow, we add four extra procedures: 1) Alongside the user enclave, the SM enclave is booted together; 2) The user enclave locally attests the SM enclave to establish a secure channel and the digest of target bitstream is securely issued to the SM enclave; 3) The key server remotely attests the SM enclave before issuing the encryption key; and 4) The expected bitstream is verified by comparing the digest, manipulated by injecting a random RoT, and encrypted within the SM enclave, isolated from any privileged attacks. The injected RoT is subsequently employed for CL attestation, which will be discussed in detail in Section 4.3.

As such, as shown in Figure 3 a new booting flow with RoT injection included consisting of three phases, device manufacturing, application development, and cloud instance deployment is designed.

Device manufacturing. A random symmetric device key, Key_{device}, is injected into every manufactured FPGA during


SM App.
(Key_{attest})
Custom Logic w/ SM Logic
(Key'_{attest})
• Generate Random Nonce N
• $MAC_{req} = MAC_{Key_{attest}}(N, DNA)$
$\langle N, MAC_{req} \rangle$
• $MAC'_{req} = MAC_{Key'_{attest}}(N, DNA')$
• Compare MAC_{req} and MAC'_{req}
• $MAC_{rsp} = MAC_{Key'_{attest}}(N+1, DNA')$
MAC_{rsp}
• $MAC'_{rsp} = MAC_{Key_{attest}}(N+1, DNA)$
• Compare MAC_{rsp} and MAC'_{rsp}

(a) Custom logic attestation.



(b) Cascaded attestation.

Figure 4. (a) CL attestation protocol. (b) Cascaded attestation of heterogeneous components.

the manufacturing process. The manufacturer also maintains a key distribution server for device-key pairs.

Heterogeneous application development. *Salus* follows a development flow similar to conventional heterogeneous applications. The developer acquires the development kit from the CSP and develops the user enclave application and the CL in a secure local environment. During the development phase, the CL developer integrates the publicly verified SM logic and generates a CL bitstream containing both the accelerator and the SM logic. The SM logic reserves a storage for the RoT, namely Key_{attest}. To manipulate the bitstream the developer records the hierarchical location of the RoT, $Loc_{Key_{attest}}$, within the generated CL netlist and stores it alongside the bitstream. After generation, the developer calculates a digest H of the bitstream and metadata. Thanks to the flexibility provided by the bitstream manipulation tool as mentioned in Section 2.3, *Salus* does not require the hierarchical location of RoT, $Lock_{Key_{attest}}$, to be fixed in a final compiled CL netlist. Consequently, the SM logic functionality can be released as a development kit and freely integrated with the accelerator logic by developers.

Cloud instance deployment. When the data owner intends to deploy a cloud instance, the user enclave application and the SM enclave application are deployed on the TEE-enabled host (①). The data owner initiates a remote attestation to verify the booted user application (②). Along with the remote attestation request, the user client transfers the metadata of the expected bitstream, $Loc_{Key_{attest}}$ and H, to the user enclave. The user enclave initiates a local attestation request to verify the SM enclave and forwards $Loc_{Key_{attest}}$ and H using a negotiated symmetric key (③). The SM enclave then requests the manufacturer for Key_{device} associated with ID of the rented FPGA, $DeviceDNA$, provided by the CSP (④). Upon receiving a key request, the manufacturer server initiates a remote attestation request to the SM enclave before distributing Key_{device}. Through this process, the SM enclave is trusted and Key_{device} is securely issued. The SM enclave fetches the CL partial bitstream, verifies its integrity by comparing H, generates a random attestation key Key_{attest}, and injects into $Loc_{Key_{attest}}$. The manipulated bitstream is then encrypted with Key_{device} and sent to the shell (⑤). All the above operations are happened within the enclave such that the manipulated plaintext bitstream is not leaked. The CSP-maintained shell forwards the encrypted bitstream to the FPGA fabric. Inside the FPGA internal fabric, the decryption logic decrypts the bitstream and writes it to the FPGA configuration memory to load CL on the dynamic area(⑥). The decryption process is inaccessible to any programmable logic and is trusted as mentioned in Section 2.3. Once the CL is loaded, the SM enclave issues an attestation request to verify integrity of the CL (⑦). The user enclave generates a remote attestation report and sends it to the data owner (⑧). Details of the CL attestation (⑦) and the user enclave remote attestation report generation (⑧) will be discussed in Section 4.3 and Section 4.4.

4.3 Custom Logic Attestation

CL attestation is to prove that both the CL and the SM enclave posses the same secret which is dynamically injected during the booting. ShEF [42] proposes a remote attestation analogous to the traditional remote attestation used in Intel SGX. The remote attestation is based on PKE which is compute-intensive and it requires a public-key infrastructure, *e.g.*, CA. In addition, the attestation is less efficient since requiring the network to authenticate certificates.

We claim that remote attestation is unnecessary under the usage of RoT injection proposed in Section 4.2. The secure CL booting flow guarantees that the attestation key is securely generated inside the SM enclave, and only a correctly loaded CL bitstream contains the key. Hence, compared to the PKE based remote attestation, we design an attestation mechanism similar to Intel SGX local attestation, relying on a symmetric key, as shown in Section 1. Similar to the SGX local attestation, the proposed CL attestation is carried over

Table 2. Analogy Between *Salus* CL Attestation And Intel SGX Local Attestation

Intel SGX Local Attestation	***Salus* CL Attestation**
Verifier enclave generates a challenge MRENCLAVE.	SM enclave generates a challenge N.
Prover enclave gets report key.	SM logic gets attestation key.
Prover enclave generates a MAC over MRENCLAVE.	SM logic generates a MAC over $N + 1$.
Prover enclave sends report containing MAC to verifier enclave.	SM logic sends report containing MAC to SM enclave.
Verifier enclave fetches local report key.	SM enclave fetches locally generated attestation key.
Verifier enclave verifies MAC with report key and MRENCLAVE.	SM enclave verifies MAC with attestation key and $N + 1$.

a unsecure channel, and it is resistant to all confidentiality attacks, integrity attacks, and freshness attacks.

A successful attestation verifies two facts: 1) Key_{attest} used in the SM logic is authenticated; and 2) The SM logic itself is authenticated. Failure of either fact results in the failure of Key_{attest} attestation. FPGA partial reconfiguration completely overwrites the reconfigurable partition, preventing attackers from partially modifying the accelerator logic while leaving the SM logic untouched to create a legitimate bitstream. Therefore, a successful Key_{attest} attestation indicates a successful authentication of the entire CL.

The detailed CL attestation protocol is described in in Figure 4a. The analogy between the CL attestation and the Intel SGX local attestation is shown in Table 2. After loading a CL, the SM enclave generates a random nonce N and computes a MAC MAC_{req} over both N and $DeviceDNA$. MAC_{req} ensures integrity of the attestation request. Upon receiving a request, the SM logic on the FPGA side uses the local Key'_{attest} and $DevcieDNA'$ to verify the MAC. Checking $DeviceDNA$ ensures that the FPGA ID assigned by the CSP matches the one used by the user-rented FPGA, confirming correctness of the FPGA. The SM logic generates an attestation response by calculating a function over the nonce and calculating a MAC, MAC_{rsp}, which is then verified by the SM enclave. As the security strength of MAC relies on the key independent of the message, for simplicity, we choose an incremental operation as the function. For security concern, the function can also apply more complicated routines, such as hashing.

4.4 Cascaded Attestation

4.4.1 Cascaded Attestation. To ensure proper attestation of a cloud FPGA instance, all heterogeneous components must go through attestation. SGX-FPGA [40] sequentially attests the user application, the SM application, and finally the CL. The cloud customer only interacts with the user application and receives the attestation report for the user enclave. The CL attestation result is not forwarded to the user end. This renders the attestation report useless in that even if the data owner receives the report, the platform is still not trusted as the CL is not attested yet at the time of receiving the report.

To address this, we propose a new attestation scheme named cascaded attestation. The cascaded attestation defers the user enclave's remote attestation report generation until the CL attestation is completed. As shown in Figure 4b, the user client initiates a remote attestation request to the user enclave, triggering a local attestation request from the user enclave to verify integrity of the SM enclave. The SM enclave generates a local attestation report and initiates the CL attestation. Once the CL attestation is completed, the SM enclave generates a message conveying the CL attestation result to the user enclave. The user enclave then generates a remote attestation report containing both CL and SM enclave verification information and responds to the user client. Within the cascaded attestation, the attestation report of each stage contains attestation results of all the backward stages, such that the attestation results of all heterogeneous components are chained together. Tampering with any stage will cause the final report sent to the user end invalid. Consequently, as soon as the data owner receives the attestation report, the data owner could immediately upload sensitive data as all the components are trusted.

4.4.2 Security Evaluation. Firstly the security of the user enclave is identified since it is remotely attested at the end of the cascaded attestation. Knowing the user enclave will be verified in the end, the successful local attestation between the two enclaves indicates the integrity of the SM enclave. Similarly, the authenticated SM enclave indicates the authentication of the CL. In summary, the local attestation and CL attestation bundle the user enclave, SM enclave, and CL together, enabling a single remote attestation from the user client to attest the entire heterogeneous platform. The cascaded attestation achieves the same level of security as the remote attestation of the standalone CPU TEE.

4.5 User Enclave Interface

To utilize this heterogeneous architecture, a secure channel is needed between the host and FPGA. *Salus* provides a secure channel for register transactions between the SM enclave and SM logic, utilizing two secrets, session key $Key_{session}$ and session counter $Ctr_{session}$, additionally injected alongside the attestation key during bitstream manipulation. For a register transaction, the user enclave first transfers a register transaction to the SM enclave via a local attestation established secure channel. The SM enclave forwards the transaction to the SM logic through a channel protected by $Key_{session}$ and

$Ctr_{session}$. The SM logic transparently decrypts, verifies, and forwards the register transaction to the accelerator. *Salus* also establishes a direct but unsecure connection between the user enclave and the accelerator bypassing the SM enclave and SM logic. It is the developer's responsibility to decide how to use the two interfaces, *e.g.*, a symmetric data key is exchanged over the secure register interface while subsequent encrypted memory transactions are carried over the direct unsecure interface.

4.6 Security Analysis

As recommended by Kerckhoff's doctrine [25], a secure system should rely only on secret keys while all the routines could be publicly known to attackers. *Salus* does not rely on any black-box implementation as the attestation key is dynamically generated and injected per deployment while all other components can be open-sourced. As mentioned in Section 3.1, *Salus* mainly focuses on two attack types, *i.e.*, integrity attacks on CL during booting and bus attacks on host-PCIe transactions, including both CL attestation transactions and runtime user data transactions.

As Table 3 shows, steps ①-⑥ guarantee the integrity of the CL during the loading process. Specifically, remote attestation and local attestation of the user enclave and the SM enclave (①,②,③) guarantee the bitstream metadata and digest are securely transferred from the user client to the SM enclave. Remote attestation of the SM enclave (④) guarantees that the device key is securely issued to the SM enclave ensuring the confidentiality and integrity. Bitstream verification using the digest within the SM enclave (④) verifies the integrity of the target CL bitstream. Bitstream manipulation and encryption within the SM enclave (④) and the CL loading (⑤,⑥) guarantee confidentiality of the injected attestation key. The above process guarantees that a verifier-only-known secret, Key_{attest}, is confidentially embedded into the target CL bitstream without leaking the secret to the shell. Any integrity attack will lose the secret and consequently fail the following symmetric key based integrity verification.

As only the SM enclave and the target CL have access to attestation key which is randomly generated per deployment, the two components establish a secure attestation channel. As such, the attestation (⑥) is resistant to PCIe bus attacks, including confidentiality, integrity, and replay attacks. Similar to the attestation key, the secure embedding of the session key guarantees that runtime data transactions are also resistant to PCIe bus attacks.

The above secure key distribution process and the assumption of the secure key management service hosted by the trusted hardware manufacturer, as mentioned in Section 3, provide secure channels to faithfully distribute symmetric keys, Key_{device} and Key_{attest}, between hardware components. This approach provides an equivalent way to establish an agreed symmetric key as the key exchange phase commonly used in asymmetric encryption protocols. The usage of symmetric cryptographic algorithms simplifies the SM logic's design complexity, reduces hardware resource consumption, and also shortens booting time in comparison to asymmetric cryptographic algorithms.

4.7 Multiple Partial Reconfiguration Partitions

In *Salus*, we currently target architectures with only one reconfiguration partition (RP). To extend to support multiple RPs, each RP is required to integrate an SM logic such that each RP can be separately programmed and attested. A more optimized solution could consist of a single master SM logic and several light-weight slave SM agents which are integrated with each RP. We leave this to our future work.

5 Implementation

We provide *Salus* implementation details in this section. Section 5.1 describes hardware design of the SM logic. Section 5.2 describes software stack of the user enclave application and host enclave application. *Salus* is not bound to low-level implementations and developers are free to choose different techniques as long as fulfilling required functionalities.

5.1 Hardware Architecture

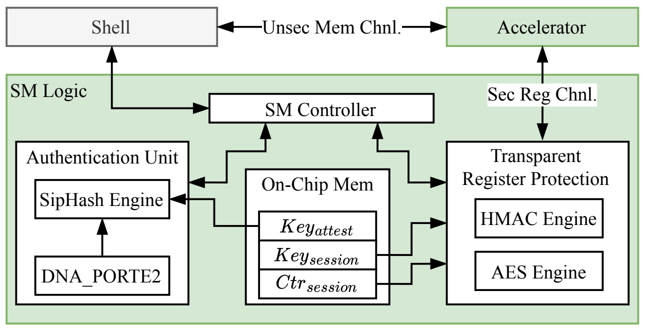


Figure 5. FPGA architecture of *Salus*.

As shown in Figure 5, all the secure CL booting related functionalities are wrapped within the SM logic module, and the module is exposed as a portable AXI4-Lite IP to connect to the shell. The SM logic exposes a AXI4-Lite interface to the accelerator for secure register transactions. The SM logic transparently guarantees confidentiality and integrity of the accelerator's register interface. All interfaces are the same for the accelerator as original unsecure FaaS, as such *Salus* puts no design limit on the accelerator. Similarly, *Salus* also puts no design limit on the shell as long as the shell can load an encrypted CL partial bitstream. The shell used in existing FaaS, such as Alibaba F3 [10] and AWS F1 [1], already supports bitstream encryption. The SM logic exposes an AXI4-Lite control interface and an AXI4 memory interface to the accelerator same as current FaaS. The developer only

Table 3. Protection of Secrets in Secure CL Booting Flow

Steps	Operation	Target Secret	C[1]	I[2]	Flow of Target Secret	Description
①②	Remote Attest.	H, $Loc_{Key_{attest}}$	✓	✓	User Client → User Enclave	Securely transfer H and $Loc_{Key_{attest}}$.
③	Local Attest.	H, $Loc_{Key_{attest}}$	✓	✓	User Enclave → SM Enclave	Securely transfer H and $Loc_{Key_{attest}}$.
④	Remote Attest.	Key_{device}	✓	✓	Manufacturer → SM Enclave	Securely transfer Key_{device}.
④	Bit. Verification	Bitstream	-	✓	SM Enclave	Fetched bitstream is verified with H.
④	Bit. Manipulation	Key_{attest}	✓	-	SM Enclave	Key_{attest} is securely injected to $Loc_{Key_{attest}}$.
④	Bit. Encryption	Key_{attest}	✓	-	SM Enclave	Key_{attest} is securely encrypted by Key_{device}.
⑤⑥	CL Loading	Key_{attest}	✓	-	SM Enclave → SM Logic	Key_{attest} is securely transferred to CL.
⑦	CL Attestation	Key_{attest}	-	✓	SM Enclave → SM Logic	Key_{attest} is verified.

[1] Confidentiality [2] Integrity

needs to integrate the SM logic by connecting the accelerator to the reserved interfaces.

5.1.1 Secure Manager Logic. As shown in Figure 5, all secrets, including Key_{attest}, $Key_{session}$, and $Ctr_{session}$, are stored in an isolated on-chip block memory (BRAM). The memory interface is not exposed outside the SM logic, such that a shell could not steal the secrets. The BRAM initial values are changed during bitstream manipulation. The FPGA ID *DeviceDNA* is retrieved using a DNA_PORTE2 IP [12]. MAC is calculated by a SipHash engine, a light-weight add-rotate-xor based pseudorandom function generating a short 64-bit MAC. SipHash guarantees that an attacker knowing a message x and MAC $SipHash(x, k)$ but not key k could not derive any message $y \neq x$ with the same MAC. In *Salus*, Key_{attest} and $Key_{sesseion}$ are not accessible by attackers, and hence the SipHash-based MAC is secure.

5.1.2 Disabling ICAP Readback. To prevent potential security risks, it is crucial for the manufacturer to disable the readback capability of the existing ICAP. Otherwise, the shell could snoop and reverse-engineer the CL, extract secrets from the CL, and counterfeit a valid authentication. On existing FPGAs, this is an inevitable security weakness from which all previous FPGA TEE works suffer [22, 31, 40, 42]. This disabling is the only feature we utilize but is not currently available on COTS FPGAs. However, it is feasible for the manufacturer to release a new ICAP IP since it is essentially an officially-encrypted piece of RTL code. Considering the potential demand in the cloud privacy market, providing such a new IP would be advantageous for the manufacturer. FPGA IDE (*e.g.*, Vivado) version control can be used such that only new IDE versions containing a new manufacturer-released readback-disabled ICAP IP can be used to develop the shell. This is more related to software supply chain security and software life cycle management and we will investigate this in our future work.

5.2 Software Stack

Salus designs two independent software stacks: one for the user client and manufacturer server, and the other for user and SM applications, as Figure 6 shows. The two software stacks result from different security assumptions and functionality requirements. The user client and manufacturer server assume a trusted environment and are only responsible for transferring encrypted data to a verified enclave. User and SM applications, on the contrary, target to cooperate to securely boot a CL bitstream on an untrusted cloud instance. *Salus* leverages gRPC remote procedure call (RPC) library [11] for easy development and extension.

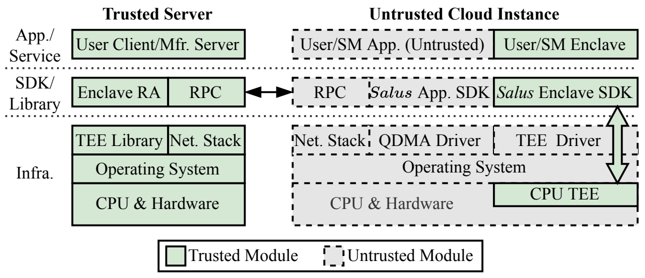


Figure 6. *Salus* software stack.

5.2.1 User Client and Manufacturer Server. The user client and manufacturer server extend the trust boundary from trusted parties to enclaves running on a cloud instance through a standard CPU TEE remote attestation process, as shown in ② and ④ in Figure 3. During remote attestation, the user/SM enclave generates an asymmetric key pair and issues the user client/manufacturer server the public key and its digest carried by an Intel SGX DCAP quote. Note that this implementation is not mandatory, as developers are free to switch to other TEE platforms and attestation mechanisms or exchange the key differently, *e.g.*, DHKE [29] [36].

After verifying the enclave, the user client/manufacturer server encrypts and transfers sensitive data to the user application/SM application, and the latter decrypts the data within the user enclave/SM enclave. For the manufacturer server, it encrypts Key_{device} before transmission to avoid key leakage (④). For the user client, the data owner sends metadata of the expected CL bitstream (H and $Loc_{Key_{attest}}$) to the user enclave (②) and securely provides Key_{data} to the

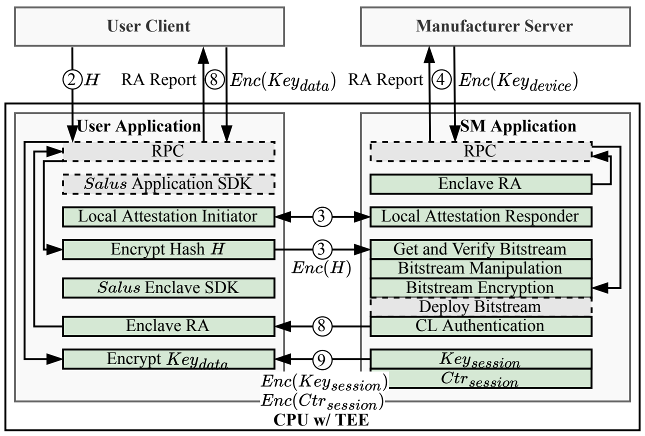


Figure 7. Detailed modules of user and SM applications.

user enclave following a successful remote attestation of the entire heterogeneous platform (⑧).

5.2.2 Cloud Instance Applications. In a nutshell, the user and SM applications cooperatively fulfill their responsibilities through following steps: 1) performing local attestation to bridge the user and SM domain; 2) verifying the CL bitstream against the shared digest H to ensure the expected bitstream is loaded; 3) inserting Key_{attest}, $Key_{session}$, and $Ctr_{session}$ using bitstream manipulation and encrypting the bitstream to establish the foundation for CL bitstream integrity; 4) deploying the encrypted bitstream; and 5) performing CL attestation to finally extend the trust from the host to the FPGA side. As Figure 7 shows, all secure booting related modules are placed within the enclave while leaving RPC modules outside for networking.

Local attestation. Using local attestation, the user enclave ensures the SM enclave is trustworthy and establishes a secure communication channel to the verified SM enclave. The two enclaves exchange a symmetric key using Elliptic-Curve Diffie-Hellman (ECDH) [18].

Bitstream operation. To ensure CL bitstream integrity, the SM application wraps bitstream verification, bitstream manipulation, and bitstream encryption as an integral operation within the enclave. During deployment, the SM application only switches out of the enclave for PCIe accesses (⑤), and continues CL attestation (⑦) within the enclave.

6 Experimental Evaluation

6.1 Experimental Setup

We conduct an evaluation of *Salus* prototype on a realistic setting. We run both the user and SM applications on a physical machine, equipped with two SGX-enabled Intel Xeon Ice Lake CPUs and 1TiB memory as well as a Xilinx Alveo U200 FPGA board. We host the manufacturer server on an Alibaba elastic compute service (ECS) instance (ecs.c7t.4xlarge), equipped with 16 vCPUs and 32GiB memory. The user client runs on a local laptop. We use an Alibaba hosted DCAP server to verify Intel SGX attestation reports.

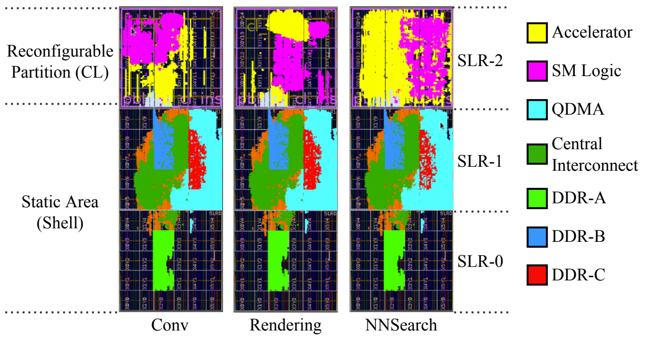


Figure 8. Floor planning of shell and CL on FPGA.

This setup simulates a real cloud scenario where a FaaS platform, a manufacturer server, and a user client are deployed in three different domains. In a real cloud usage, the user client runs securely on the user side and connects to a remote heterogeneous cloud instance which is hosted by a CSP. The manufacturer server is securely deployed and maintained on the manufacturer's side, as an independent secure key distribution service remotely serving all the CSPs. Our experiment setup accurately simulates remote connections between the three entities.

Xilinx's open-source RapidWright [13] offers a Java package for bitstream manipulation. We host it by Occlum [35], a library OS running on an enclave. Since Xilinx does not provide an open-source bitstream encryption tool, we implement an AES-GCM-256 routine within the enclave to encrypt plaintext bitstreams to measure bitstream encryption time. The encryption algorithm aligns with the one used in Vivado according to an official document [17].

We develop a light-weight shell supporting bitstream encryption. Figure 8 shows the floor planning of our design. All logics are developed using Vivado 2022.1. To evaluate FPGA TEE runtime performance, we customize five open-source accelerators to operate on ciphertext user data, by adding memory encryption/decryption logic and integrate with the SM logic. Table 4 lists the benchmarking applications.

6.2 Resource Utilization

The resource utilization of various accelerators and the SM logic are listed in Table 5. Despite different benchmark applications, the total available CL resource is fixed as it is only determined by the area reserved for RP during floor planning stage. In the implementation, we reserved one super logic region as the RP, occupying approximately one-third of the FPGA resources, only for prototyping purpose. The functionality of the SM logic is general such that the resource utilization of the SM logic is the same across all the benchmarks. Among the available CL hardware resources, the SM logic only consumes 8% LUTs, 4% Registers, and 13% BRAMs.

Table 4. Benchmarking Applications

Application	Description	Source	Added Memory Encryption
Conv	Single convolution layer over a 3×3×256 kernel.	Xilinx SDAccel Example [14]	Input feature maps.
Affine	Affine transformation on a 512×512 image.	Xilinx SDAccel Example	Input & output images.
Rendering	Render 2D images from 3D models.	Rosetta [43]	Input & output images.
FaceDetect	Viola Janes face detection [37].	Rosetta	Input image.
NNSearch	Nearest-neighbour linear search.	Xilinx SDAccel Example	Input targets and queries.

Table 5. Resource Utilization Breakdown of CL

Logic	LUT	Register	BRAM
Total CL Resource	355 040	710 080	696
Conv	19 735 (6%)	20 169 (3%)	329 (47%)
Affine	32 014 (9%)	36 382 (5%)	543 (78%)
Rendering	29 132 (8%)	35 731 (5%)	142 (20%)
FaceDetect	31 956 (9%)	36 201 (5%)	62 (9%)
NNSearch	49 069 (14%)	42 568 (6%)	122 (18%)
SM Logic	**27 667 (8%)**	**29 631 (4%)**	**88 (13%)**

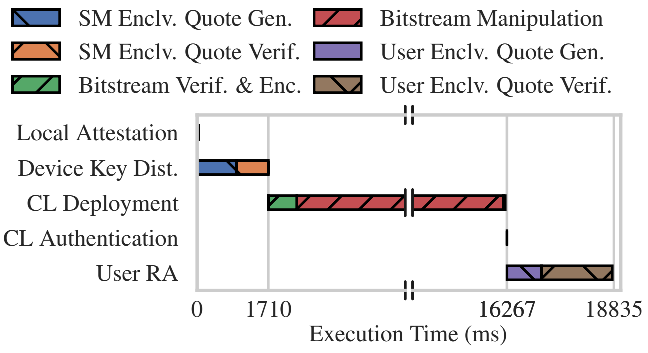


Figure 9. Execution time of CL booting.

The lightweight SM logic enables easy code inspection and portability for release. As shown in Figure 8, the SM logic can be freely integrated by the developer such that the location of the SM logic and consequently $Loc_{Key_{attest}}$ are dynamic across different compiled CL netlists.

6.3 Booting Time

Figure 9 shows that booting and attesting a CL takes extra 18.8 seconds on top of 40+ seconds booting time of a CSP VM instance [6, 16]. As the booting time is a one-shot overhead, we consider this booting time reasonable. As a comparison, the booting process in ShEF [42] consumes 5.1 seconds without implementing and measuring network handshaking.

The bitstream manipulation during the CL bitstream deployment consumes most of the total booting time (73.2%). Directly wrapping the RapidWright inside an enclave without tailoring results in an inefficient implementation. The bitstream verification and encryption takes 725 milliseconds in total. It takes 1709 milliseconds for the manufacturer to remotely attest and distribute a key to the SM enclave. The remote attestation of the user enclave takes 2568 milliseconds. The user client running on a laptop connects to the DCAP server through a wide-area network, which explains why it takes longer than on the manufacturer server, which connects through an intra-cloud network. Requiring no network communication, local and CL attestations are negligible (836 microseconds and 1.3 milliseconds, respectively).

We need to note that the time of bitstream verification, manipulation, and encryption, is the same for all applications. More specifically, the time of bitstream operations is only dependent on the size of the partial CL bitstream. For Xilinx devices and toolsets, a partial CL bitstream's size is only determined by the area reserved for the CL during floor planning, while independent of accelerator logic implemented inside the reconfigurable area [3]. In our experiments, we reserve the same region for all CL applications such that the bitstream operation time is the same.

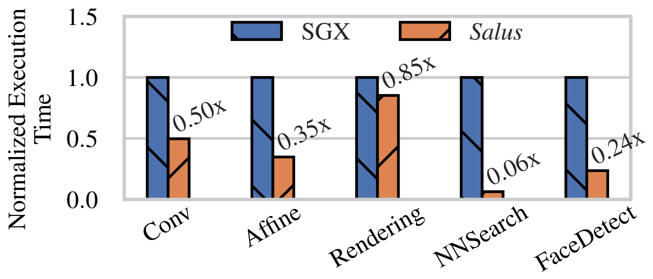


Figure 10. Performance of realistic workloads running on a securely booted FPGA TEE.

6.4 Performance of Realistic Workloads

We run five realistic workloads on an isolated FPGA TEE to compare performance benefits to running on a CPU TEE. We add an AES-CTR streaming encryption/decryption logic at the memory interface. Depending on the application, we selectively encrypt memory transactions. For machine learning tasks, **Conv**, **FaceDetect**, and **NNSearch**, we only encrypt incoming traffic while leaving training weights and outputs in plaintext. For **Affine** and **Rendering**, we encrypt both inbound and outbound traffic. Baseline implementations run solely on an SGX-enabled CPU.

Table 6. Slowdown of CPU TEE And FPGA TEE

Implementation	Conv	Rendering	FaceDetect
CPU w/o TEE	3038.52 ms	1.24 ms	26.69 ms
CPU w/ TEE	3059.90 ms	5.43 ms	93.38 ms
CPU Slowdown	**1.01×**	**4.38×**	**3.50×**
FPGA w/o TEE	1522.09 ms	4.40 ms[1]	21.50 ms[1]
FPGA w/ TEE	1522.20 ms	4.63 ms	22.05 ms
FPGA Slowdown	**1.00×**	**1.05×**	**1.03×**

[1] Measured and reported on Xilinx U200 FPGAs by Rosetta [43].

As shown in Figure 10, *Salus* achieves a speedup of 1.17 to 15.64 times over SGX. To further analyze the root cause of the speedup, we compare performance of the plaintext non-SGX CPU implementation, the ciphertext SGX CPU implementation with added memory encryption/decryption, the plaintext non-TEE FPGA implementation, and the ciphertext FPGA TEE implementation with added memory encryption/decryption, as shown in Table 6. The overhead of the FPGA TEE is negligible when comparing the accelerator performance running within and outside the FPGA TEE. Tested on **Conv**, **NNSearch**, and **FaceDetect**, the TEE-protected accelerator runs only up to 1.01 times slower than a non-protected accelerator. This negligible overhead results from the high-throughput memory traffic encryption within the accelerator. In addition, the accelerator logic stores intermediate data on internal on-chip block memories without writing to the external memory, further reducing memory traffic and intermediate memory encryption overhead. In contrast, the CPU TEE overhead is as high as 4.38 times. The data movement between trusted and untrusted contexts is encrypted by a general OpenSSL-based encryption. Additionally, all memory accesses within the enclave program , such as *malloc*, are forced to be transparently encrypted by hardware, resulting in extra overhead.

The requirement for the memory traffic encryption gives a larger design space during the accelerator design and possibly higher speedup over CPU implementations. For an application where the plaintext FPGA implementation already achieves some speedup over the plaintext CPU implementation, the performance gap might be enlarged when running on TEEs and operating on encrypted sensitive data justifying the significance of an FPGA TEE.

7 Related Works

FPGA TEE. In addition to SGX-FPGA [40], ShEF [42], Ambassy [22], and MeetGo [31], which are discussed in Section 3.2, there are also other TEE works targeting accelerators. HETEE [44] implements a centralized security controller on an FPGA placed in a tamper-resistant box, targeting a rack-level isolation, which is orthogonal to the board-level isolation studied in this work. CRONUS [24] introduces fault isolation between spatially shared accelerators, which is not considered in this work, as COTS FaaS does not host multiple accelerators simultaneously. GuardNN [20] targets ML accelerators instead of a general cloud infrastructure.

GPU TEE. Graviton [39] designs a light-weight peripheral hardware added to a GPU and a runtime running on a host TEE to isolate sensitive kernels from other code and device drivers. LITE [41] introduces a light-weight customizable software encryption scheme. HIX [23] refactors the GPU driver code to run within a host TEE to avoid modifications to the existing GPU architecture, while requiring a change on the CPU-GPU I/O interconnect. Due to the fundamental architectural difference, these approaches cannot be directly applied to FPGAs.

PCIe 6.0 specification introduces TEE Device Interface Security Protocol (TDISP) [15], a new architecture to secure I/O virtualization. The standardized interface enables a secure key exchange between a VM and a PCIe device. However, TDISP does not handle a potentially malware CSP-maintained shell. The key exchange used in *Salus* can be standardized by TDISP, and we leave this to our future work.

8 Conclusion

As the volume of sensitive data increases, a trusted and high-performance CPU-FPGA heterogeneous cloud infrastructure is highly demanded. *Salus* addresses this need by building a TEE on off-the-shelf cloud FPGA devices. Targeting heterogeneous architectures, *Salus* assumes a TEE-enabled CPU on the host and extends the CPU TEE boundary from the host to the accelerator.

Acknowledgments

We thank our shepherd, Tamara Lehman, and the anonymous reviewers for their insightful feedback that much improved this paper. We also appreciate AMD for donating an Alveo U200 accelerator card and the Alibaba Cloud ECS team for their technical support.

References

[1] Amazon ec2 f1 instances. https://aws.amazon.com/ec2/instance-types/f1/, 2024.
[2] Amd adaptive computing documentation portal. https://docs.xilinx.com/r/en-US/ug909-vivado-partial-reconfiguration, 2024.
[3] Amd partial reconfiguration. https://docs.amd.com/v/u/QIPlm0yVRKL~grcqwY2VWQ, 2024.
[4] Attestation services for intel® software guard extensions. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html, 2024.
[5] Azure confidential computing | microsoft learn. https://learn.microsoft.com/en-us/azure/confidential-computing/, 2024.
[6] Comparing the speed of vm creation and ssh access of cloud providers. https://blog.cloud66.com/part-2-comparing-the-speed-of-vm-creation-and-ssh-access-on-aws-digitalocean-linode-vexxhost-google-cloud-rackspace-packet-cloud-a-and-microsoft-azure, 2024.
[7] Confidential computing capabilities. https://www.alibabacloud.com/help/en/ecs/user-guide/confidential-computing-capabilities/, 2024.

[8] Deploy ml models to fpgas - azure machine learning | microsoft learn. https://learn.microsoft.com/en-us/azure/machine-learning/v1/how-to-deploy-fpga-web-service?view=azureml-api-1#fpga-support-in-azure, 2024.

[9] familykey • vitis unified software platform documentation: Embedded software development (ug1400) • reader • amd adaptive computing documentation portal. https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/familykey, 2024.

[10] Fpga as a service. https://www.alibabacloud.com/help/en/fpga-based-ecs-instance, 2024.

[11] grpc. https://grpc.io/, 2024.

[12] Introduction • ultrascale architecture libraries guide (ug974) • reader • documentation portal. https://docs.xilinx.com/r/en-US/ug974-vivado-ultrascale-libraries, 2024.

[13] Rapidwright documentation — rapidwright 2022.2.0-beta documentation. https://www.rapidwright.io/docs/index.html, 2024.

[14] Sdaccel_examples/getting_started/clk_freq/large_loop_ocl at master · xilinx/sdaccel_examples. https://github.com/Xilinx/SDAccel_Examples/tree/master/getting_started/clk_freq/large_loop_ocl, 2024.

[15] Tee device interface security protocol (tdisp) | pci-sig. https://pcisig.com/tee-device-interface-security-protocol-tdisp, 2024.

[16] Understanding and profiling gce cold-boot time | by colt mcanlis | google cloud - community | medium. https://medium.com/google-cloud/understanding-and-profiling-gce-cold-boot-time-32c209fe86ab, 2024.

[17] Using encryption and authentication to secure an ultrascale/ultrascale+ fpga bitstream application note. https://www.xilinx.com/content/dam/xilinx/support/documents/application_notes/xapp1267-encryp-efuse-program.pdf, 2024.

[18] Elaine Barker, Lily Chen, Sharon Keller, Allen Roginsky, Apostol Vassilev, and Richard Davis. Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography. Technical report, National Institute of Standards and Technology, 2017.

[19] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[20] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G Edward Suh. Guardnn: Secure accelerator architecture for privacy-preserving deep learning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 349–354, 2022.

[21] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud {GPUs}. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 817–833, 2020.

[22] Dongil Hwang, Sanzhar Yeleuov, Jiwon Seo, Minu Chung, Hyungon Moon, and Yunheung Paek. Ambassy: A runtime framework to delegate trusted applications in an arm/fpga hybrid system. *IEEE Transactions on Mobile Computing*, 2021.

[23] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 455–468, 2019.

[24] Jianyu Jiang, Ji Qi, Tianxiang Shen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Gong Zhang, Xiapu Luo, and Heming Cui. Cronus: Fault-isolated, secure and high-performance heterogeneous computing for trusted execution environment. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 124–143. IEEE, 2022.

[25] Auguste Kerckhoffs. *La cryptographie militaire, ou, Des chiffres usités en temps de guerre: avec un nouveau procédé de déchiffrement applicable aux systèmes à double clef.* Librairie militaire de L. Baudoin, 1883.

[26] Michał Kucab, Piotr Boryło, and Piotr Chołda. Hardware-assisted static and runtime attestation for cloud deployments. *IEEE Transactions on Cloud Computing*, 2023.

[27] Chris Lavin and Alireza Kaviani. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–140. IEEE, 2018.

[28] Kristiyan Manev, Joseph Powell, Kaspar Matas, and Dirk Koch. byteman: A bitstream manipulation framework. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–9. IEEE, 2022.

[29] Ralph C Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[30] Mathias Morbitzer, Benedikt Kopf, and Philipp Zieris. Guarantee: Introducing control-flow attestation for trusted execution environments. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 547–553. IEEE, 2023.

[31] Hyunyoung Oh, Kevin Nam, Seongil Jeon, Yeongpil Cho, and Yunheung Paek. Meetgo: A trusted execution environment for remote applications on fpga. *IEEE Access*, 9:51313–51324, 2021.

[32] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting {SGX} enclaves from practical {Side-Channel} attacks. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 227–240, 2018.

[33] Rakin Muhammad Shadab, Yu Zou, Sanjay Gandham, Amro Awad, and Mingjie Lin. Hmt: A hardware-centric hybrid bonsai merkle tree algorithm for high-performance authentication. *ACM Transactions on Embedded Computing Systems*, 22(4):1–28, 2023.

[34] Rakin Muhammad Shadab, Yu Zou, Sanjay Gandham, and Mingjie Lin. Omt: A run-time adaptive architectural framework for bonsai merkle tree-based secure authentication with embedded heterogeneous memory. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 191–202. IEEE, 2023.

[35] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, 2020.

[36] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 708–721, 2020.

[37] Nitish Srivastava, Steve Dai, Rajit Manohar, and Zhiru Zhang. Accelerating face detection on programmable soc using c-based synthesis. In *25th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb 2017.

[38] Flavio Toffalini, Eleonora Losiouk, Andrea Biondo, Jianying Zhou, and Mauro Conti. {ScaRR}: Scalable runtime remote attestation for complex systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 121–134, 2019.

[39] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *OSDI*, pages 681–696, 2018.

[40] Ke Xia, Yukui Luo, Xiaolin Xu, and Sheng Wei. Sgx-fpga: Trusted execution environment for cpu-fpga heterogeneous architecture. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 301–306. IEEE, 2021.

[41] Ardhi Wiratama Baskara Yudha, Jake Meyer, Shougang Yuan, Huiyang Zhou, and Yan Solihin. Lite: A low-cost practical inter-operable gpu tee. In *Proceedings of the 36th ACM International Conference on Supercomputing*, pages 1–13, 2022.

[42] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. Shef: Shielded enclaves for cloud fpgas. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1070–1085, 2022.

[43] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018.

[44] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, et al. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1450–1465. IEEE, 2020.

[45] Yu Zou, Amro Awad, and Mingjie Lin. Hermes: Hardware-efficient speculative dataflow architecture for bonsai merkle tree-based memory authentication. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 203–213. IEEE, 2021.

[46] Yu Zou and Mingjie Lin. Fast: A frequency-aware skewed merkle tree for fpga-secured embedded systems. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 326–331. IEEE, 2019.