ShEF: Shielded Enclaves for Cloud FPGAs

Mark Zhao
myzhao@cs.stanford.edu
Stanford University
Stanford, California, USA

Mingyu Gao
gaomy@tsinghua.edu.cn
Tsinghua University
Beijing, China

Christos Kozyrakis
christos@cs.stanford.edu
Stanford University
Stanford, California, USA

ABSTRACT

FPGAs are now used in public clouds to accelerate a wide range of applications, including many that operate on sensitive data such as financial and medical records. We present *ShEF*, a trusted execution environment (TEE) for cloud-based reconfigurable accelerators. *ShEF* is independent from CPU-based TEEs and allows secure execution under a threat model where the adversary can control all software running on the CPU connected to the FPGA, has physical access to the FPGA, and can compromise the FPGA interface logic of the cloud provider. *ShEF* provides a *secure boot and remote attestation* process that relies solely on existing FPGA mechanisms for root of trust. It also includes a *Shield* component that provides secure access to data while the accelerator is in use. The Shield is highly customizable and extensible, allowing users to craft a bespoke security solution that fits their accelerator's memory access patterns, bandwidth, and security requirements at minimum performance and area overheads. We describe a prototype implementation of *ShEF* for existing cloud FPGAs, map *ShEF* to a performant and secure storage application, and measure the performance benefits of customizable security using five additional accelerators.

CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; **Cloud computing**; • **Security and privacy** → **Hardware security implementation**.

KEYWORDS

trusted execution, enclaves, cloud computing, FPGAs, reconfigurable computing

ACM Reference Format:
Mark Zhao, Mingyu Gao, and Christos Kozyrakis. 2022. *ShEF*: Shielded Enclaves for Cloud FPGAs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507733

1 INTRODUCTION

Cloud computing is a double-edged sword. Cloud servers provide unmatched capabilities that are highly available, easily deployable, and widely scalable. This flexibility is critical for data-driven applications such as machine learning. However, the massive volume of data flowing through shared infrastructure presents new security and privacy concerns due to the corresponding growth in the trusted computing base (TCB) of cloud applications. When users process sensitive data on the cloud, they implicitly trust a multitude of entities including the developers and operators of the underlying hardware, OSes/VMMs, storage and database systems, and identity and access management services. They also trust the employees of the cloud service provider (CSP) who have physical access to the infrastructure. Recent data leaks demonstrate that a vulnerability in any layer of the stack can result in exposure of highly-sensitive information, e.g., the health and financial records for hundreds of millions of people [5, 9, 66].

These issues have led to the development of software security mechanisms to protect sensitive data in the cloud. Users can use purely cryptographic solutions such as homomorphic encryption (HE) [46] or integrate ad-hoc cryptographic schemes from a plethora of libraries. Unfortunately, HE is prohibitively expensive for most real-world applications [27]. Traditional cryptographic libraries, assuming correctness, still depend on a large TCB, including the CSP's multiple layers of software controlled by potentially malicious employees.

CPU-based trusted execution environments (TEEs), such as Intel SGX [33] and ARM TrustZone [17] shrink the TCB. TEEs provide hardware-based isolation for user code and data even given malicious privileged software and physical attacks. However, hardware is inherently *hardened*, presenting many security problems. First, cryptography and cryptanalysis are constantly evolving; standards and best-practices constantly improve, especially as new computing paradigms such as quantum computing are introduced [26]. Second, applications use numerous compute and communication patterns, each demanding different levels of protection. For example, some applications may only require authenticated encryption for streaming data, while others read and write multiple times to a given address and thus need additional security such as Merkle Trees [85]. Finally, the recent vulnerabilities that directly compromise SGX highlight the difficulty of implementing bug-free security mechanisms in modern CPUs [28, 30, 58, 61, 65, 71, 80, 83, 91–93]. For these reasons, it is desirable to have the flexibility to enhance security mechanisms post-manufacturing, a trait that hardened CPU security mechanisms do not provide.

Moreover, CPU-based TEEs do not directly enable isolated execution on accelerators such as GPUs [31], FPGAs [45, 95], or TPUs [55]. Slowing trends in process scaling are driving specialized hardware in order to achieve scalable performance [50]. As a result, CSPs including Amazon [2], Microsoft [8], Huawei [11], and Baidu [13], are rapidly deploying remote FPGAs to meet increased computing

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507733

demands, allowing users to deploy custom accelerators generated from application code [32, 48, 59, 69, 81].

Supporting secure computing on remote FPGAs is thus essential for many emerging applications, such as medical, financial, machine learning (ML), and regulatory use-cases, that require both security and acceleration. For example, FPGAs are at the forefront for accelerating genomics sequencing [89, 98]. FPGAs are also extensively used for accelerating Deep Neural Networks (DNNs) [82], which have demonstrated significant impact in medical diagnosis [40]. Recent work even directly call for accelerator TEEs, including FPGAs, as an essential component of distributed federated learning in the cloud [101] and GDPR-compliant storage systems [53]. These sensitive accelerators are seeing widespread adoption in industry, with Xilinx marketing FPGAs directly for ML-based diagnostics [21] and myriad companies offering AWS F1 accelerators for genomics [7, 14, 15], financial analysis [10], and network security [12] applications on the AWS Marketplace.

Unfortunately, recently-proposed TEEs for accelerators, including FPGAs, are either insecure against direct physical attacks [54, 105], require fundamental hardware changes [54, 94, 105], only address isolated challenges such as attestation [36, 52, 72, 99], or rely on external CPU TEEs [52, 54, 94]. Moreover, they ignore the Shell logic [3, 57, 60, 88], a fundamental untrusted operating system for cloud FPGA logic.

We address these challenges with the *Shielded Enclaves for Cloud FPGAs* (*ShEF*) framework that brings together hardware-based *bespoke security* and *customizable acceleration* for FPGAs. *ShEF* applies the confidentiality, integrity, freshness, and isolation guarantees provided by TEEs [78] to custom FPGA accelerators, even in the presence of malicious software and hardware logic, or physical attacks. *ShEF* targets current cloud FPGA deployments and commodity FPGA hardware. While *ShEF* relies on CPUs for networking and data transfer, it is decoupled from and assumes no trust in CPU TEEs or other software running on CPUs. Hence, *ShEF* minimizes the TCB and avoids the aforementioned issues with *hardened* hardware. *ShEF* provides *customization* as a key feature, enabling users to adapt security mechanisms to match their accelerator's unique bandwidth requirements, memory access characteristics, and threat models. By provisioning only the right levels of protection, *ShEF* allows users to address their threat model at minimum performance and area cost.

ShEF consists of two main components. The *ShEF boot process* centers around a software security kernel that builds a chain of trust on top of currently-existing FPGA security mechanisms. Its primary purpose is to (*a*) load the accelerator into a known and trusted state on the FPGA, (*b*) attest the state to a remote verifier *chosen by the user*, and (*c*) ensure that sensitive ports such as JTAG are secured during runtime.

Once the accelerator design is securely booted onto the FPGA, the *ShEF Shield* communicates with host software and protects the accelerator's sensitive data through a highly customizable and extensible set of soft-logic engines. Users can customize a rich set of parameters, such as encryption logic parallelism, optimizations for memory access patterns, cryptographic primitives, authentication block size, and key size over individual regions of memory. For example, a DNN accelerator that reads in large blocks of weights can choose to use large encryption blocks to amortize integrity check overheads and forgo expensive replay-attack countermeasures specifically for those weights. Other accelerators that perform multiple small reads and writes (e.g., for graph processing) may conversely select smaller block sizes to prevent unnecessary data transfers and use heavyweight memory authentication techniques [37].

In summary, we make the following contributions:

- We identify requirements on enabling TEEs on cloud FPGAs and gaps not addressed by prior work due to current cloud FPGA devices and environments.
- We implement an end-to-end workflow on **current** FPGA devices that provides the first comprehensive and customizable TEE for cloud FPGAs.
- We present protocols that enable important TEE building blocks, including secure boot and remote attestation, on top of our workflow.
- We identify customizability as a key requirement for cloud FPGAs and provide a solution via the modular Shield to easily customize *ShEF* to the diverse security and performance requirements of FPGA-based accelerators.

We implemented the end-to-end *ShEF* workflow on a representative FPGA, and we configured the Shield's parallelism and security for six accelerators with diverse performance requirements and access patterns, including a GDPR secure storage benchmark [53] and DNNWeaver [81]. We demonstrated that *ShEF* minimized overheads to 0-122% with 3.1-11% area on AWS F1 instances. *ShEF* is open-source[1], allowing the community to review and build on its design.

2 BACKGROUND AND MOTIVATION

ShEF is motivated by the confluence of two important trends [49]. First, hardware security is becoming a first-class citizen, driven by the ongoing microarchitectural side-channel attacks to CPUs [28, 58, 61, 65, 80] and SGX in particular [30, 71, 83, 91–93]. Second, the end of Dennard scaling is galvanizing a shift towards energy-efficient domain-specific accelerators (DSAs). It is unclear how to bridge the gap and provide secure computation for DSAs which have disparate security and performance requirements.

ShEF realizes recent requests for FPGA TEEs in order to provide secure remote storage [53]. However, *ShEF* goes further and makes the key insight to leverage the flexibility of cloud FPGAs to enable secure remote acceleration. Numerous applications such as machine learning [48], genomics [98], multi-party computation [97], and simulation [56] already utilize DSAs on cloud FPGAs. The goal of *ShEF* is to provide security guarantees with the same level of flexibility and to create bespoke TEEs for cloud FPGAs that fit accelerators' specific needs. TEEs, FPGA security, and cloud FPGAs are all well-studied domains. Merging these domains, however, introduces the new challenges discussed in this section.

2.1 Trusted Execution Environments (TEEs)

TEEs secure remote computation by providing an isolated environment for users' sensitive code on devices controlled by untrusted parties and running untrusted, privileged software.

[1] https://github.com/stanford-mast/ShEF

While there exist numerous flavors of TEEs [17, 29, 33, 34, 41, 42, 64, 67, 74, 85–87], there are necessary and sufficient building blocks that they all provide [78]. TEEs are built on a chain of trust, starting with a **hardware-based root of trust** in the form of a private key stored securely on chip [78, 103]. A **secure boot** process extends trust by cryptographically measuring each component during boot, up to and including the secured application. This integrity measurement is then cryptographically proven to a remote user of the secured application in a **remote attestation** process. Once trust is established, confidential data is emplaced into the TEE from a **secure storage and I/O** channel. The secure application processes the data, and the TEE ensures that any interaction with the rest of the system is secured via an **isolated execution** mechanism.

In the context of spatial computing platforms, such as FPGAs, we assert that a TEE must also be **customizable**. Accelerators use a selection of I/O interfaces, exhibit a broad range of memory access and throughput characteristics, and require different levels of security mechanisms. A static TEE that provides a general solution for all accelerators is doomed to either be over-instrumented and waste resources or not satisfy each accelerator's stringent throughput and security requirements.

2.2 Conventional FPGA Security Mechanisms

Xilinx [20] and Intel [16] FPGAs target mission-critical applications such as defense and networking, and thus adopt a similar threat model and security mechanisms [24, 76]. Namely, a single bitstream developer must have physical and secure access to the FPGA before deploying the device into an untrusted environment. The security mechanisms' goal is to ensure that (a) only developer-signed bitstreams can be loaded, (b) bitstreams are encrypted to prevent reverse-engineering, and (c) the FPGA can detect and respond to physical tampering. These mechanisms are enabled in Intel and Xilinx FPGAs via a series of redundant, embedded processor modules executing from BootROM and programmable firmware [24, 76]. We refer to these as the Security Processor Block (SPB) hereinafter.

The SPB has access to two pieces of information embedded in secure, on-chip, non-volatile storage: an AES key and the hash of a *public* ECDSA (Intel) or RSA (Xilinx) key. The AES key can be further encrypted via a physically-unclonable function (PUF), preventing the AES key from being compromised under physical attacks. The device owner or IP developer is meant to securely embed the necessary keys offline prior to deployment. Depending on whether the developer desires encryption and/or authentication, she can encrypt the bitstream with the AES key and/or sign it with the ECDSA/RSA private key. The bitstream can then be securely decrypted (using the AES key) and authenticated (using the public key hash) by the SPB. Finally, the SPB actively monitors for any tampering.

2.3 Remote FPGAs-as-a-Service

FPGAs are becoming an increasingly important component for major cloud service providers (CSPs) [2, 8, 11, 13]. The most prevalent example is AWS EC2 F1 instances, which provide 1, 2, or 8 exclusive Xilinx Virtex UltraScale+ VU9P FPGAs, each with 64GB of local DDR4 memory, tethered to a host CPU via a dedicated PCIe x16 link.

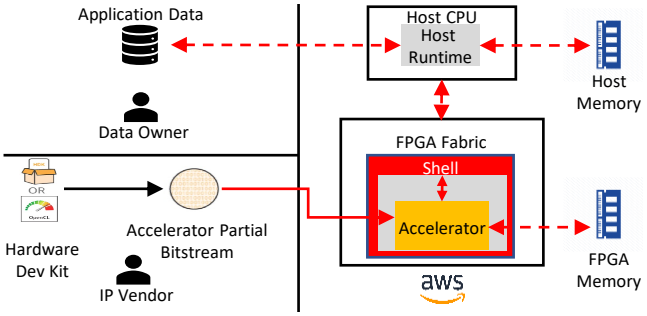

Figure 1: The AWS F1 development process and corresponding *ShEF* threat model. The red arrows represent untrusted channels.

F1 instances are currently offered in a traditional Infrastructure-as-a-Service (IaaS) fashion as shown in Figure 1. IP vendors use the AWS FPGA Development Kit [3] to create a custom accelerator using Xilinx's development tools such as RTL, OpenCL, or C/C++ High-level Synthesis (HLS). Once the accelerator design is finalized, the developers compile it into a bitstream binary. AWS leverages partial reconfiguration, which allows disjoint spatial regions of the FPGA fabric to be programmed via separate partial bitstreams. An F1 instance is configured with two partial bitstreams: one belonging to the CSP which contains the *Shell* logic, and one belonging to the user's accelerator design. The Shell is analogous to an operating system as it provides the accelerator with virtualized peripherals and debugging features (virtual JTAG/LEDs), while protecting the physical FPGA from malicious logic [88]. The Shell is static logic and continuously runs on the FPGA. At design time, developers connect their accelerator module's I/O ports to the standard Shell interfaces. At deployment time, users leverage a command line interface to dynamically program their chosen partial bitstream onto the remaining reconfigurable region. Once an accelerator is programmed, a runtime program on the host CPU initiates the accelerator and facilitates execution by transferring data between the CPU memory and FPGA device memory.

2.4 Challenges for Secure and Customized Computing

ShEF aims to enable secure and customized computing on cloud FPGAs by providing key TEE building blocks. However, fundamental assumptions made by FPGA manufacturers and inherent differences between FPGA and CPU cloud offerings induce important challenges not addressed by prior work.

A lack of asymmetric keys. As described in Section 2.2, FPGA manufacturers assume a threat model for a single, trusted user whose AES key and public key hash are loaded onto the FPGA in a secure facility. In contrast, cloud TEEs need to be utilized by multiple, mutually distrusting parties who never have physical access to the FPGA. *Thus, an FPGA TEE must build a hardware root-of-trust and remote attestation protocol on top of the available AES key, as opposed to traditional private keys assumed by CPU and prior accelerator TEEs [33, 51, 54, 78, 94, 105].*

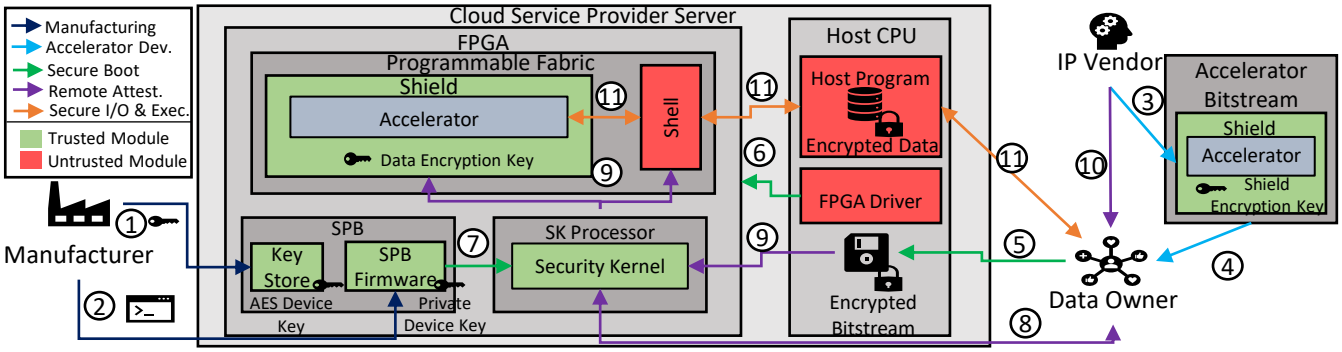

Figure 2: Architecture of the *ShEF* workflow with color-coded legend.

Presence of an untrusted Shell. Furthermore, many mechanisms CPUs use to enable TEE building blocks do not apply to the spatial architecture of FPGAs. In CPUs, threads belonging to both secure and insecure applications are time-sliced on the processor. Thus, enclaves can directly access secure hardware using ISA extensions [33], bypassing the untrusted OS. For example, the SGX ISA extensions allow users to directly access hardware mechanisms to boot an enclave (ECREATE, EADD, EXTEND, and EINIT), generate an attestation report and provision secrets (EREPORT), and provide isolated execution (EENTER and EEXIT, and SGX MEE [47]). Similarly, Keystone [64] relies on a security monitor running in the RISC-V machine mode with control over ISA-defined Physical Memory Protection registers in order to enforce memory isolation. In cloud FPGAs, however, the fabric is spatially shared with the persistent and untrusted Shell logic. *Any and all I/O ports are untrusted*, as applications' custom logic can only connect to a series of I/O ports defined and exposed by the Shell.

Lack of secure and flexible I/O. Many FPGA-based accelerators operate on data-intensive problems and stress off-chip I/O with unique security requirements. For example, some accelerators (e.g., a Bitcoin miner) only require secure register-level access, while other accelerators may exhibit more complex memory accesses including streaming (e.g., for DNN weights) and random-accesses (e.g., for graph processing). As we explore in Section 5, each accelerator requires distinct security and performance levels. Current work on FPGA security [25, 35, 36, 52, 72, 99] does not address the lack of secure I/O as a result of the Shell, nor do they provide security mechanisms that can adapt to the distinct security and performance levels required by diverse accelerators.

2.5 Threat Model

We assume the comprehensive threat model in Figure 1. The adversary attempts to compromise the confidentiality and integrity of the code and data processed by the user's accelerator running on the FPGA. The adversary has *full physical control of the FPGA device post-manufacturing* and is *able to control privileged CPU software*, such as the OS, VMM, and device drivers. Furthermore, the adversary is *able to control privileged FPGA logic, such as the AWS F1 Shell*. We assume that *any off-chip memory, including HBM, can be compromised*, as the adversary can either perform physical attacks on off-chip buses, or intercept traffic via the Shell logic for HBM. We assume that the physical package, supply chain, and manufacturer of the FPGA are trusted. While we use a host program to transfer data, we assume *the host CPU is untrusted* and do not depend on any security mechanisms provided by the CPU TEEs. By removing the host CPU from the TCB, we are not susceptible to the recent attacks that plague CPU TEEs [28, 30, 58, 61, 65, 71, 80, 83, 91–93].

Side-channel attacks are largely a function of applications' specific logic. Thus, we do not claim defend against all possible side-channel attacks. Instead, *ShEF*'s customizability allows us to provide established tools for developers to mitigate common FPGA side-channel attacks such as controlled channel [100] and power analysis [79, 102] attacks (see Section 5.2). We hope the open-source community will leverage *ShEF*'s flexibility to contribute additional security mechanisms.

We do not consider denial-of-service (DoS) attacks, as the CSP has physical control of the hardware and can simply unpower it. The CSP is incentivized to prevent DoS attacks due to the revenue loss when the FPGA instance is unavailable. We do not consider attacks against the CSP, as the Shell already protects against malicious FPGA users. We do not consider covert channels; we assume the tool flow used to generate the accelerator is trusted and run in a secure environment.

3 *SHEF* WORKFLOW AND SECURITY MODEL

ShEF is an end-to-end framework that enables remote users to design bespoke TEEs for accelerators in cloud FPGAs. *ShEF* is designed around existing mechanisms in cloud FPGAs (i.e., Xilinx UltraScale+ and Intel Stratix 10) and does not require hardware changes. That said, *ShEF* does necessitate the cooperation of both FPGA manufacturers and CSPs to realize the TEE requirements described in Section 2.1. This section provides an overview of *ShEF* and its components, as well as the responsibility of all parties. Section 4 describes how these components enable the requisite TEE building blocks.

The four key parties in the *ShEF* framework are shown in Figure 2. The **Manufacturer** is responsible for manufacturing the physical FPGA chip. The **Cloud Service Provider (CSP)** owns the physical datacenter that houses servers containing FPGAs and offers them to customers. *ShEF* splits the notion of a "customer" into two separate entities. The **IP Vendor** creates the actual accelerator

design and distributes it to **Data Owners** (e.g., on a public marketplace [4]). The Data Owner then rents an FPGA instance from the CSP, programs the accelerator, and uses it to process sensitive data. The Data Owner should source accelerator designs only from trusted IP Vendors. Of course, the IP Vendor and Data Owner can be the same entity. The Data Owner does not have to trust the CSP, but does assume trust in the Manufacturer and the IP Vendor.

We now review each party's role and how trust is delegated by following the steps in the *ShEF* framework in Figure 2.

Device Manufacturing (the Manufacturer). The security foundation of *ShEF* begins with the Manufacturer. The Manufacturer must provision two keys for each FPGA device: an AES device key and an asymmetric public/private device key pair (e.g., RSA or ECDSA). The Manufacturer must burn the AES device key into an e-fuse or BBRAM (and optionally encrypt it with a PUF) using existing FPGA security mechanisms ① during production. The Manufacturer then embeds the asymmetric private device key into the FPGA SPB firmware and then encrypts the firmware using the AES device key ②. The Manufacturer must also register and publish the public device key via a trusted certificate authority.

Accelerator Development (the IP Vendor). The IP Vendor is trusted to develop the accelerator IP in a secure environment such as a secure workstation ③. During the development process, the IP Vendor connects the accelerator's I/O ports to the open-source *ShEF Shield* module instead of the Shell interface. The *Shield* provides configurable runtime secure I/O and isolated execution (see Section 5). Since the Shield is parameterized RTL logic exposing the same interface as the Shell, the IP Vendor can easily simulate and integrate her design with the Shield. The IP Vendor can secure multiple accelerator modules with separate Shield modules, enabling multiple isolated execution environments [57].

The IP Vendor then provisions a symmetric Bitstream Encryption Key and an asymmetric Shield Encryption Key for the accelerator. The Bitstream Encryption Key is used for bitstream confidentiality, and the Shield Encryption Key is used to protect data transferred between the Data Owner and the FPGA (both explained below). The IP Vendor embeds the private Shield Encryption Key into each respective Shield module and compiles the entire design into a partial bitstream. Finally, the IP Vendor encrypts the partial bitstream with the Bitstream Encryption Key and distributes the encrypted partial bitstream ④. The IP Vendor creates one accelerator bitstream for all users; the attestation process provisions unique keys to each Data Owner.

Deployment (the Data Owner). Once the Data Owner is ready to process sensitive data using the accelerator, she obtains a cloud FPGA instance from the CSP ⑤. The Data Owner then instructs the CSP's FPGA driver to program the accelerator onto the FPGA. The FPGA driver first resets the FPGA, which initiates a secure boot process ⑥. The SPB begins by using BootROM code to load the SPB firmware from disk, decrypting it using its embedded AES device key. The SPB firmware boots the *ShEF* Security Kernel from external storage onto a dedicated Security Kernel Processor executing from its own private, on-chip memory ⑦. The Security Kernel Processor can either be a reserved hardened CPU in the FPGA or a static bitstream containing a soft CPU (e.g., MicroBlaze or Nios II) [76]. The SPB firmware hashes the Security Kernel and uses its hash and the private device key to generate a unique asymmetric Attestation Key pair and a corresponding certificate. As a result, the Attestation Key is cryptographically bound to the FPGA device and the specific Security Kernel binary. The Security Kernel itself contains no confidential information and has no direct access to device keys, preventing attackers from leaking the device keys via an illegitimate Security Kernel. The Security Kernel only has access to the Attestation Keys that it receives from the SPB firmware via a secure channel (e.g., on-chip shared memory).

The *ShEF* Security Kernel has three primary jobs. First, it performs remote attestation with the Data Owner and IP Vendor ⑧ (Section 4). Through this process, the Data Owner receives cryptographic proofs showing that the FPGA device, the Security Kernel, and the accelerator partial bitstream are authentic, referencing the Manufacturer and the IP Vendor as certificate authorities, respectively. Second, it mediates all access to the FPGA fabric. The CSP uses the Security Kernel to first launch the Shell into its reserved static logic region. The Security Kernel then securely receives the accelerator's Bitstream Encryption Key from the IP Vendor via a secure channel established during attestation, and decrypts and loads the accelerator onto the FPGA, connecting it to the Shell interface via partial reconfiguration ⑨. Since the Security Kernel is open source and contains no secrets, the CSP can fully control and audit the Shell loading process. Likewise, the Security Kernel hash is included in the attestation report (Section 4); the IP Vendor audits the hash before sending over the Bitstream Encryption Key. Finally, the kernel continuously checks existing hardware monitors. It can thus detect backdoor activity (e.g., JTAG and programming ports) [24, 76], ensure that the authenticated accelerator bitstream is not modified before use, and prevent any physical attacks. While the Security Kernel relies on the host CPU to communicate with the IP Vendor, this channel is authenticated and encrypted.

As part of the remote attestation process, the Data Owner generates a symmetric Data Encryption Key for each Shield module (Section 4). Data Encryption Keys are used to encrypt the sensitive input data. The Data Owner receives the IP Vendor's public Shield Encryption Key ⑩, and encrypts the Data Encryption Key(s) against the IP Vendor's public Shield Encryption Key to produce Load Key(s). The Load Key(s) are subsequently used to securely provision the Data Encryption Key into each *ShEF* Shield module.

Finally, the Data Owner is ready to utilize the accelerator, using an (untrusted) *ShEF* host program on the untrusted host CPU to proxy all communication to the accelerator ⑪. The host program forwards the Load Key and the encrypted data to the FPGA. The *ShEF* Shield uses the private Shield Encryption Key to decrypt the Load Key(s) and retrieve the Data Encryption Key(s), which in turn secures the user data during runtime. When the outputs are ready, the Shield transfers results, encrypted using the Data Encryption Key, back to the Data Owner via the host program. As is the case with the Security Kernel, all communication through the CPU is encrypted and authenticated.

4 SECURELY ENABLING TEE BUILDING BLOCKS

We now present a security argument to demonstrate how *ShEF* enables all of the TEE building blocks (Section 2.1).

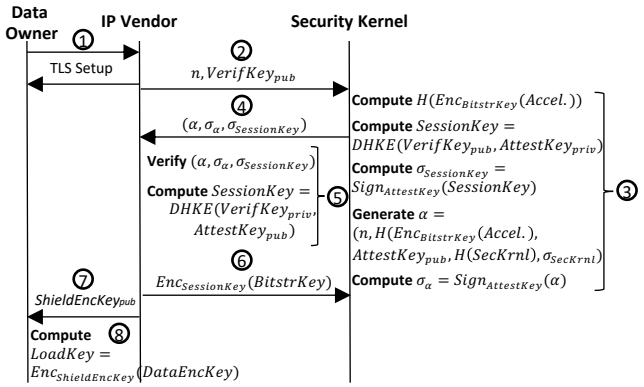

Figure 3: Remote Attestation and Secure Storage and I/O Protocols.

Hardware Root-of-Trust. While current FPGAs do not provide hardware support for requisite private asymmetric keys, *ShEF* is able to build a root-of-trust via two manufacturer-provisioned keys. The AES device key is the true root-of-trust, protected by existing mission-critical security mechanisms in current FPGAs. The private device key provides the asymmetric cryptography needed for attestation. Although it is embedded in firmware, it is encrypted with the AES device key and thus imbued with the same level of trust.

Secure Boot. *ShEF*'s secure boot process is built as an extension to current FPGA boot mechanisms, which first executes BootROM code on the SPB. The BootROM decrypts the SPB firmware using the AES device key and hands off the boot process to it. We trust that this step is secure as it is deployed in numerous mission-critical FPGA applications.

Once the SPB firmware is initialized, its main job is to bootstrap trust to the Security Kernel running on a dedicated processor. To do so, it reads the Security Kernel out of the boot medium and hashes it to obtain $H(\texttt{SecKrnl})$. The SPB firmware signs the hash with the private device Key $\texttt{DeviceKey}_{\texttt{priv}}$. It uses the resulting value to seed a key generator to produce a unique asymmetric Attestation Key pair $\texttt{AttestKey}_{\texttt{priv,pub}}$, which is cryptographically bound to the device and the Security Kernel binary. It also generates a certificate over the Security Kernel and the resultant Attestation Key by computing $\sigma_{\texttt{SecKrnl}} = \text{Sign}_{\texttt{DeviceKey}}(H(\texttt{SecKrnl}), \texttt{AttestKey}_{\texttt{pub}})$. The SPB firmware then loads the Security Kernel onto the processor and places the Attestation Key pair and $\sigma_{\texttt{SecKrnl}}$ into the Security Kernel's private memory. Secure boot occurs before any other software is loaded, ensuring that no untrusted software can tamper with or monitor the Security Kernel. If the Security Kernel Processor is a soft CPU, its partial bitstream is hashed alongside the Security Kernel and the bitstream is loaded by the SPB firmware.

Remote Attestation. Once the Security Kernel boots, it waits for a remote attestation request. Remote attestation executes via a series of message exchanges between the Data Owner, the IP Vendor, and the Security Kernel, shown in Figure 3. Through remote attestation, (a) the Data Owner generates an ephemeral Data Encryption Key used to secure sensitive data, (b) the IP Vendor validates the authenticity of the FPGA device and bitstream, and (c) the Security Kernel receives the Bitstream Key required to load the accelerator. Data Owners are in full control of which IP Vendor to use for remote attestation for each FPGA instance.

The Data Owner initializes a standard TLS/SSL connection to a trusted IP Vendor server ①. The IP Vendor first generates a random nonce n, as well as an asymmetric Verification Key pair $\texttt{VerifKey}_{\texttt{priv,pub}}$. The IP Vendor sends n and $\texttt{VerifKey}_{\texttt{pub}}$ to the Security Kernel ②.

Meanwhile ③, the Security Kernel reads and hashes the appropriate encrypted accelerator bitstream, obtaining $H(\text{Enc}_{\texttt{BitstrKey}}(\texttt{Accelerator}))$. Using $\texttt{VerifKey}_{\texttt{pub}}$ and $\texttt{AttestKey}_{\texttt{priv}}$, the Security Kernel performs key exchange to generate a shared symmetric SessionKey with the IP Vendor, allowing the Security Kernel and the IP Vendor to send encrypted messages. To prevent any man-in-the-middle attacks, the Security Kernel also signs SessionKey with $\texttt{AttestKey}_{\texttt{priv}}$ to obtain a new certificate $\sigma_{\texttt{SessionKey}}$. The Security Kernel then generates an attestation report α, containing n, $H(\text{Enc}_{\texttt{BitstrKey}}(\texttt{Accelerator}))$, $\texttt{AttestKey}_{\texttt{pub}}$, $H(\texttt{SecKrnl})$, and $\sigma_{\texttt{SecKrnl}}$. The Security Kernel signs this report using $\texttt{AttestKey}_{\texttt{priv}}$ to obtain σ_α, and finally sends back α, σ_α, and $\sigma_{\texttt{SessionKey}}$ to the IP Vendor ④.

The IP Vendor authenticates the attestation report starting with the $\texttt{DeviceKey}_{\texttt{pub}}$ received through the Manufacturer's certificate authority ⑤. The IP Vendor checks that $\sigma_{\texttt{SecKrnl}}$ was signed by the corresponding $\texttt{DeviceKey}_{\texttt{priv}}$, proving that a legitimate FPGA generated the attestation report. To ensure that the Security Kernel (and Security Kernel Processor, if applicable) is valid, the IP Vendor consults a public list of *ShEF* Security Kernel (and Security Kernel Processor) hashes. Next, the IP Vendor authenticates the attestation report by using $\texttt{AttestKey}_{\texttt{pub}}$ to ensure that α was signed with $\texttt{AttestKey}_{\texttt{priv}}$. The IP Vendor matches the signed nonce with n, preventing replay attacks, and the signed bitstream hash with $H(\text{Enc}_{\texttt{BitstrKey}}(\texttt{Accelerator}))$, confirming that the correct bitstream was loaded into the Security Kernel's memory. Finally, the IP Vendor establishes a secure channel to the Security Kernel by first generating the same SessionKey as the Security Kernel using $\texttt{AttestKey}_{\texttt{pub}}$ and $\texttt{VerifKey}_{\texttt{priv}}$, and verifying that $\sigma_{\texttt{SessionKey}}$ was signed by $\texttt{AttestKey}_{\texttt{priv}}$.

Using the SessionKey, the IP Vendor securely transmits the BitstrKey to the Security Kernel ⑥. The Security Kernel decrypts the accelerator bitstream and loads it onto the FPGA, ensuring that the plaintext bitstream containing sensitive IP and Shield Keys are only handled in secure on-chip memory.

Secure Storage and I/O. *ShEF* provides secure storage and I/O to the Data Owner by creating a security perimeter via the *ShEF* Shield that encrypts and authenticates all data external to it. Recall that the IP Vendor provisioned a private Shield Encryption Key into each Shield module in the accelerator. As part of the remote attestation session, the IP Vendor provides the public Shield Encryption Key to the Data Owner (e.g., via a certificate authority) ⑦. The Data Owner generates at least one Data Encryption Key (e.g., one per Shield module) and encrypts them against the public Shield Encryption Key to get the Load Key(s) ⑧. The Data Owner then encrypts sensitive input data in a secure location using the appropriate Data Encryption Key. The Load Key(s) are later sent to the FPGA Shield, which decrypts it to get the Data Encryption

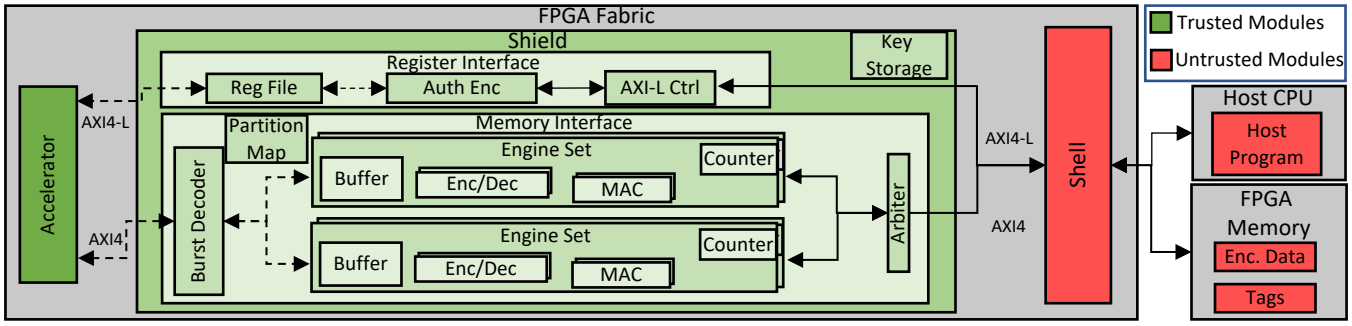

Figure 4: *ShEF* Shield architecture. Dashed and solid lines correspond to unencrypted and encrypted data, respectively.

Key(s). The Shield uses it to ensure that all sensitive data are secured during both storage and I/O as we discuss in Section 5.

Isolated Execution. The Shield and Security Kernel ensure isolated execution. The Security Kernel constantly runs and monitors the FPGA programming and debug ports to prevent tampering with accelerator logic during execution.

5 *SHEF* SHIELD: SUPPORT FOR FLEXIBLE SECURITY

The Shield is a highly-configurable RTL module that provides isolated execution and secure I/O and storage by interposing on ports between the accelerator and Shell. The Shield also realizes the necessary *customization* for FPGA TEEs. IP Vendors configure the Shield to fit their accelerators' memory access patterns and performance and security requirements.

5.1 Shield Overview

Figure 4 shows how the Shield fits within a typical cloud FPGA deployment consisting of a host CPU and an FPGA accelerator. The host program coordinates execution via standard device drivers. It first loads the Load Key onto the FPGA, which the Shield decrypts into the Data Encryption Key and stores in ephemeral key storage. The host program then proxies encrypted commands and data between the Data Owner and Shield via the CSP's Shell interfaces. The host program is untrusted and does not observe any unencrypted data.

The CSP's Shell logic provides two primary interfaces to the host program and the accelerator. An AXI4-Lite interface, mastered by the Shell, exposes memory-mapped registers for commands and small amounts of data to the host. The accelerator and host drive an AXI4 and DMA interface, respectively, to access FPGA device memory through the Shell. While we focus on securing device memory, Shells commonly provide a generic AXI4 interface for both memory and PCIe [3]. Thus, the Shield can also support additional interfaces such as PCIe via the same AXI4 interface.

The Shield provides a wrapper module that transparently secures these interfaces. The host program accesses registers via AXI4-Lite as before, and the accelerator accesses device memory via the same AXI4 protocol. The Shield transparently decrypts and encrypts I/O between the host program, the accelerator, and the device memory as shown in Figure 4. Thus, IP Vendors can simply incorporate and configure the Shield at design time in a plug-and-play manner. Cryptographic modules that provide authenticated encryption are at the core of the Shield. We use AES-CTR + HMAC modules as default and present configurable alternatives in Section 6.2.4.

Register interface. The register interface provides authenticated encryption using the Data Owner's Data Encryption Key. The host program memory-maps accelerator-accessible registers and reads/writes encrypted data and commands via pointers. For writes from the host program, the Shield decrypts and authenticates the data before storing into the accelerator's plaintext register. Similarly, when the host program reads a mapped address, the corresponding plaintext register is encrypted and tagged before sent to the Shell. The Shield provides a register file as shown in Figure 4; users may optionally use their own register file with the Shield simply decrypting/authenticating the AXI4-Lite interface. Furthermore, because AXI4-Lite register addresses may reveal sensitive information, the Shield offers an additional option of encrypting both addresses and data via a common address for all registers. In this case, the Shield will decrypt and proxy the data to/from the appropriate plaintext register.

5.2 A Flexible Memory Interface

5.2.1 Why Flexibility Is Necessary. As shown in Figure 4, the Shield exposes the same set of AXI4 interfaces as the Shell. While the interface is generic, accelerators follow diverse paradigms when using device memory. Some accelerators stream in large blocks of data from memory, perform random accesses in on-chip buffers, and stream out results to a separate memory region. For example, deep neural networks (DNNs) stream in large blocks of weights. In applications like graph processing, accelerators use non-sequential data-dependent memory accesses. Even a single accelerator can exhibit distinct paradigms across memory regions, e.g., large streaming reads for weights and smaller reads/writes for activations in a DNN.

The accelerator diversity, in terms of the number of reads/writes to a memory region as well as the amount of data transferred in each burst, have both performance and security implications for the Shield. First, authenticated encryption, the Shield's core security mechanism, needs to be performed over variable chunk sizes. Smaller chunks require more read requests with higher compute and storage overheads for integrity tags, while overly large chunks transfer unnecessary data bytes. It is important to correctly size the chunk granularity for each accelerator's memory regions.

Second, accelerators that read and write the same chunk of memory yield an additional vulnerability. Computing a MAC over the chunk and its address prevents *spoofing attacks* that directly modify memory contents and *splicing attacks* that copy contents of memory from one address to another [37]. However, it does not preclude *replay attacks* where old values are returned for a read, since the corresponding MAC tag is valid. To safeguard against replay attacks, secure processors resort to Merkle Tree verification schemes [85], in which MACs are organized in a tree. Since SRAM is precious in CPUs, only the root node is always kept on-chip, while nodes to a leaf are validated when accessing its corresponding address.

5.2.2 Designing for Flexibility. The Shield's memory interface is designed to allow IP Vendors to configure its features and performance, enabling bespoke TEEs customized to each accelerator. The memory interface consists of one or more *engine sets*. Each burst request is transformed by a burst decoder in the Shield, which consults a map of IP Vendor-specified memory regions and maps each address range to one of the engine sets. Each engine set includes encryption and authentication engines alongside on-chip buffers and counters.

The engine set for each memory region can be *configured separately* by the IP Vendor to optimize for the different needs and threat models of the target application. We review the configuration options in the current Shield implementation:

Cryptographic engines. Each engine set contains configurable encryption (AES) and authentication (HMAC/PMAC) engines. The AES engine contains an internal 256-byte lookup table for the S-box. At design time, the S-box can be duplicated up to 16 times per engine, reducing the AES latency through parallel lookups at the cost of higher resource consumption. Users are also able to configure the AES key size (128 or 256 bits) during bitstream compilation. Since the engines expose a simple valid/ready interface, IP Vendors can simply substitute a new cryptographic engine in their place. We demonstrate this flexibility in Section 6.2.4 by substituting a PMAC engine in the place of HMAC to enable parallel MAC calculations. Thus, IP vendors can instantiate multiple instances of each engine to increase parallelism, or even use their own designs.

Chunk size. The IP Vendor denotes a chunk size C_{mem} for each memory region, which specifies the granularity of each authenticated encryption chunk. Each chunk is associated with a 12-byte initialization vector (IV), which is incremented by 1 for each successive chunk to ensure that no two ciphertext blocks reuse the same IV. Each chunk is authenticated via a 16-byte MAC tag in encrypt-then-MAC mode stored in DRAM. By using large C_{mem} values for streaming patterns, IP Vendors can better amortize the MAC tag overheads. C_{mem} can be any size, from a byte up to the entire FPGA memory.

On-chip buffers. Each engine set optionally includes a buffer, implemented using Block RAM or UltraRAM, that reduces the overheads for random accesses within small memory regions. It stores decrypted and authenticated plaintext data and their address ranges, and can be thought as a cache with a line size of C_{mem}. If a burst request hits in the buffer, the engine handles all transfers without accessing DRAM. For misses, the engine set simply generates burst requests to read the entire C_{mem}-byte chunk and 16-byte MAC tag from DRAM. The engine set decrypts and authenticates the returned ciphertext in parallel and fill in the buffer line. If the buffer needs to evict a modified line, the engine set encrypts the line and calculates a MAC tag over the ciphertext, and then performs the necessary writes to DRAM. For writes, the engine set can first fill the buffer line in the same manner as reads. Alternatively, if the corresponding chunk is only written to once and not read (e.g., streaming writes), the IP Vendor can simply zero-out the on-chip buffer, avoiding unnecessary reads.

Advanced integrity verification. If an accelerator reads and writes the same chunk multiple times, additional mechanisms are required to prevent replay attacks. The Shield supports optimized Bonsai Merkle Trees [77] that create *Merkle Trees over counters* as opposed to data chunks.

Merkle Trees are expensive for FPGA designs that need to access *every* tree node from DRAM, unlike CPUs that can benefit from multiple tiers of caches. We make the observations that Merkle Trees are used in CPU TEEs because (a) on-chip storage is scarce in CPUs, and (b) secure processors must secure relatively small cacheline-sized chunks (64B) over multiple gigabytes of memory. Accelerators, however, do not face the same issues, as (a) contemporary FPGAs provide much more on-chip memory via new technologies such as UltraRAM [19], and (b) accelerators generally operate on smaller address regions and can leverage large C_{mem} chunks. Thus, *ShEF* offers a simpler and more efficient alternative by leveraging the excess of on-chip RAM of FPGAs to store counters only over required address regions (i.e., those that read and write chunks multiple times).

Specifically, an on-chip counter module is configured for these address regions, in addition to the above encrypted authentication mechanisms. The engine set increments the on-chip counter value ctr_i by 1 for each write of chunk i to DRAM. On every read to chunk i from DRAM, a tag $\text{MAC}(i, \text{ciphertext}_i, \text{ctr}_i)$ is generated and verified against the off-chip tag. In this case, only one extra DRAM access is needed, eliminating excessive off-chip accesses associated with Merkle Trees. Since the IP Vendor can tailor C_{mem}, the counter size, and the memory region size to each specific workload, the excess storage overheads are minimized.

Side Channels. The flexibility of the Shield can also help mitigate a number of side-channel attacks discussed in Section 2.4. For controlled-channel attacks, IP vendors can significantly reduce the number of data-dependent memory accesses by increasing C_{mem} as an effective countermeasure [44, 100], increasing security by trading off bandwidth and storage efficiency. *ShEF* also provides two effective countermeasures against remote power analysis. First, *ShEF* hides the accelerator's microarchitecture via bitstream encryption, of which current power analysis attacks require significant knowledge [79, 102]. Second, *ShEF* provides a script to generate an active fence of logic that hides sensitive power signals [62].

However, certain classes of side-channel attacks, such as timing side-channel attacks, are application-dependent and require application knowledge to mitigate [90]. While we ensure that the timing of Shield cryptographic engines does not depend on any confidential information, we rely on IP Vendors to integrate application-specific techniques if timing noninterference is desired. We discussed how the register interface can be secured against address metadata attacks in Section 5.1. Further security mechanisms against address metadata attacks, such as ORAM [84], can simply be added by

adopting open-source modules (e.g., [43]) on top of Shield engines due to their generic interface.

6 IMPLEMENTATION AND EVALUATION

We implemented and evaluated *ShEF* on Xilinx UltraScale+ FPGAs. Since *ShEF* only relies on the AES key storage and an SPB, it can also be implemented on Intel FPGAs. Since we are not allowed to deploy our secure boot process on AWS, we first implemented our end-to-end workflow on a local UltraScale+ Ultra96 FPGA board [18]. We then deployed various Shield configurations with accelerators on AWS F1 instances in order to evaluate performance, assuming correct boot and attestation.

In both cases, we implemented the Shield as portable RTL code in Vivado 2019.2. The Shield interfaces with the host program via a *ShEF* runtime library, which links against the Xilinx runtime (XRT) that provides FPGA drivers and libraries. Thus, as described in Section 6.2.4, *ShEF* supports accelerators developed using RTL and frameworks such as OpenCL and SDAccel. XRT and the host program are not in the TCB.

6.1 End-to-End Ultra96 Implementation

As mentioned in Section 3, the root-of-trust is a set of two keys, one AES key embedded in the secure storage and an asymmetric key in the encrypted firmware. We provisioned an AES key into the Ultra96 e-fuses. The Ultra96 boot process first executes out of BootROM on the SPB with exclusive access to cryptographic hardware and programming ports. We embedded the private device key into the firmware and encrypted it with the AES device key. The firmware is decrypted and loaded onto another hardened processor called the platform management unit. We boot the Security Kernel on a dedicated Cortex-R5 core running solely from dedicated on-chip memory. The Security Kernel communicates with the SPB via a dedicated IPI interface to access cryptographic hardware to generate attestation reports, and runs on the R5 core continuously, monitoring programming and debug ports.

We implemented the full end-to-end *ShEF* workflow, using a Bitcoin accelerator (Section 6.2.4), on the Ultra96 board. We measured that the boot process, from power-on to bitstream loading, completes in 5.1 seconds. This is relatively small compared to the commonly-observed 40+ second boot time of CSP VM instances [22, 23], plus the approximate 6.2 seconds of bitstream loading time we observe on F1.

6.2 Shield Evaluation on AWS F1

We use AWS F1 instances to evaluate the area and performance overheads of trusted execution with the *ShEF* Shield. The *ShEF* Shield introduces encryption and integrity checks to all off-chip memory accesses. This can create bandwidth bottlenecks if the encryption/authentication rate is lower than the effective data rate of off-chip memory. This can be addressed by properly configuring the Shield in three ways: a) partitioning the address space (if possible) to use multiple engine sets, b) using multiple AES and PMAC engines within an engine set, or c) increasing AES engine S-box parallelism. This tradeoff between performance and Shield resource overheads must be carefully managed by the accelerator developer, leveraging the customizable design of *ShEF*, in order to achieve the desired security and performance at the lowest cost.

Table 1: Shield component utilization on AWS F1. The three base modules on top do not include crypto and on-chip memory (OCM).

Component	BRAM	LUT	REG
Controller	0 (0%)	2348 (0.26%)	547 (0.03%)
Engine Set	2 (0.12%)	1068 (0.12%)	2508 (0.14%)
Reg. Interface	0 (0%)	3251 (0.36%)	1902 (0.11%)
AES-4x	0 (0%)	2435 (0.27%)	2347 (0.13%)
AES-16x	0 (0%)	2898 (0.32%)	2347 (0.13%)
HMAC	0 (0%)	3926 (0.44%)	2636 (0.15%)
PMAC	0 (0%)	2545 (0.28%)	2570 (0.14%)
OCM	Variable	0 (0%)	0 (0%)

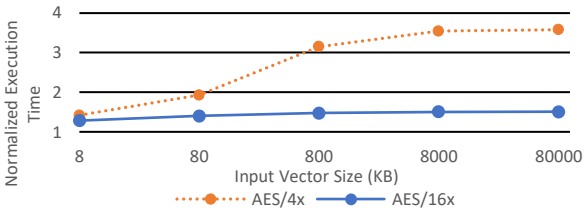

Figure 5: Vector add throughput overhead for Shield configurations.

6.2.1 FPGA Resources Overheads. Table 1 shows the FPGA resources (BRAM, LUTs, and registers) used by the Shield components. An accelerator's Shield contains one Controller, a configurable number of Engine Sets, and one Register Interface, whose base resource requirements are shown in the top of Table 1. The bottom section shows the requirements of various cryptographic engines and on-chip buffers as discussed in Section 5, which are used to augment the Register Interface and Engine Sets. For encryption, we present two types of AES engines with 4x or 16x parallelism in the S-box (Section 5). For authentication, we provide a SHA-256 HMAC engine, as well as a PMAC engine based on AES. Finally, the configurable buffers and counters use a portion of on-chip memory (max available 382Mb). A full Shield configuration's requirement is a sum of all components and their cryptographic engines and buffers which combined commonly uses single-digit percents of memory, LUTs, and registers, establishing that *ShEF* is cost effective to use with cloud FPGAs.

6.2.2 Throughput Microbenchmarks. To illustrate the Shield's parallelism, we use a vector-vector add microbenchmark that streams in two vectors and outputs their sum. The input and output vectors are partitioned and secured with four engine sets each; each set contains one AES-128 and HMAC engine and uses a 512-byte chunk. The actual logic is minimal and the workload is strictly bound by off-chip memory accesses. Figure 5 shows the relative slowdown as a function of the vector size for two Shield configurations. Slowdown is relative to the execution time without the Shield (insecure version). For short vectors, execution time is dominated by initialization overheads, e.g., data movement and signalling between the FPGA and CPU. For long vectors, the overhead directly relates to the encryption throughput available. By increasing the

Table 2: Performance overhead across Shield designs for SDP.

(# Eng. per Set S-box parall. MAC Module)	4x Eng. 4x HMAC	4x Eng. 16x HMAC	4x Eng. 16x PMAC	8x Eng. 16x PMAC	16x Eng. 16x PMAC
% Overhead	298	297	59	20	20

AES S-box parallelism, the slowdown drops below 50% for all vector sizes. We analyzed similarly a matrix multiply microbenchmark, which yielded similar, but less pronounced, insights (maximum overhead of 1.26x for AES/4x) as matrix multiplication involves more computation per data accessed.

6.2.3 End-to-End Design Example. We now demonstrate how *ShEF* can easily enable security for a real-world, end-to-end use case. SDP [53] recently proposes a GDPR-complaint storage solution by coupling distributed smart Storage Nodes (SNs), to provide encryption-at-rest and line-rate throughput, with a centralized Controller Node (CN), to control access policies and bootstrap SNs. The authors directly cite the need for an FPGA TEE at each SN. FPGAs provide both the necessary programmability to adapt to evolving access policies and line-rate throughput difficult to achieve with just CPUs. Meanwhile, the TEEs protect users' keys and data by a) attesting that the SN is authentic before provisioning keys, and b) encrypting and authenticating application and storage traffic.

SDP thus integrates FPGA TEEs as follows. On startup, a TEE within each SN remotely attests to the CN. The CN securely provisions a database of user keys into the TEE. Applications access files within SNs via TLS, providing a user-specific identity. Logic within the FPGA TEE on each SN encrypts all traffic to the application (via a TLS key) and storage device (via the user's specific key) for GDPR compliance.

Enabling SDP. *ShEF* directly enables SDP via the workflow presented in Section 3. The GDPR-compliant company assumes the IP Vendor role and creates a bitstream combining a key-value store IP, mapping user identities to files, with the Shield. SNs securely boot the bitstream and remotely attest their identity to the CN. The CN encrypts all user keys with the Data Encryption Key and securely provisions them into SN FPGA memory. Applications then directly form a TLS connection with the Shield, providing a user identity for each file access. The Shield automatically encrypts and authenticates files between the application and underlying storage. By ensuring that each SN is authenticated and securing line-rate I/O traffic, *ShEF* allows the company to deploy high-throughput SNs anywhere in the cloud.

Configuring Shield Performance. While the Shield provides automatic encryption-at-rest, it must be properly configured to meet throughput and security requirements. To demonstrate this, we created an SDP accelerator that performs gets/puts using a key-value store engine on top of the Shield. The Shield encrypts and authenticates file accesses via the user key (to storage) and the TLS key (to the application). Depending on file characteristics, the IP vendor can configure the Shield to authenticate files on a per-file or per-block basis.

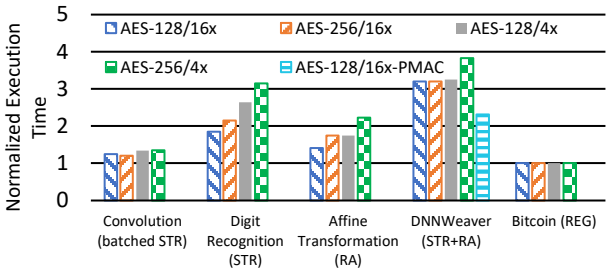

Figure 6: Execution time of workloads across Shield configurations. Value in parentheses denotes the workload's memory access characteristics (STR=streaming, RA=random accesses, REG=register). PMAC is an additional optimization for DNNWeaver.

We implemented our SDP accelerator on AWS F1 instances and compared Shield overheads to the unsecured key-value store. Table 2 shows normalized, steady-state throughput overheads across Shield configurations for 1MB file accesses, using a 4KB authentication block size. We used two identical engine sets each with 16KB buffer — one for the storage device and one for TLS. Instead of using multiple engine sets, which required complex address space partitioning due to variable-length accesses, we increased engine parallelism within each engine set by simply setting a configuration parameter in the Shield. We began with 4 AES/4x and 1 HMAC engine, but observed high overheads due to the application's high memory intensity. We attempted to increase S-box parallelism to 16x, but observed limited improvement as HMAC was the bottleneck. We replaced the HMAC engine with PMAC engines and continued to increase the number of AES/PMAC engines until performance saturated at 8x parallelism for each. Thus, the Shield's configurability allowed us to enable SDP with only 20% overhead to line-rate, and with only 4.3%, 5.0%, and 2.5% of the BRAM, LUT, and REG area, respectively.

6.2.4 Customized Engines for Various Workloads. To demonstrate how the Shield creates a bespoke TEE across accelerators with disparate memory access patterns and security requirements, we use five realistic benchmarks: a convolutional layer from a neural network with an input size of 27×27×96, a filter size of 5×5, and an output size of 27×27×256 with 32-bit values from a Xilinx reference implementation [6]; the digit recognition task from the Rosetta suite using the MNIST dataset [104]; an affine transformation kernel over 512×512 input images from a Xilinx vision accelerator [1]; DNNWeaver using LeNet [81]; and a Bitcoin mining benchmark with a hash difficulty of 24. To configure the Shield, we instantiated multiple engine sets and engines, depending on the interfaces of each application, and successively increased engine parallelism to saturate memory or accelerator bandwidth. Figure 6 shows performance results for four AES engine configurations, with either 4x or 16x S-box parallelism, and 128-bit or 256-bit keys.

Convolution achieves high parallelism by streaming in batches of feature maps and filters, and streaming out each output feature map. We configure the Shield to match the high parallelism by using 8 engine sets for input images and weights and 4 engines sets for output filters, each with one AES and HMAC engine. We

Table 3: Inclusive resource utilization on AWS F1 for the largest Shield configuration across accelerators.

Resource	Convolution	Digit Rec.	Affine	DNNWeaver	Bitcoin
BRAM	2.9%	0.71%	2.1%	3.1%	0%
LUT	11%	3.3%	11%	7.1%	1.4%
REG	5.2%	1.4%	5.2%	3.5%	0.42%

use a buffer of 128KB in the read set and 64KB in the write set. We configure C_{mem} to be 512 bytes to maximize AXI burst lengths for high throughput. In doing so, we observe small overheads of 1.20x-1.35x.

Digit Recognition also uses separate engine sets for streaming inputs and outputs, but it does not batch. Hence, we use just 2 engine sets for inputs and 1 engine set for outputs with total 24KB and 12KB buffer, respectively, each with one AES and HMAC engine. By using a large C_{mem} of 512 bytes, we are able to achieve overheads of 1.85-3.15x. We expect even lower overheads with batching by amortizing input overheads.

Affine Transformation reads non-sequential data, but reads each address once with no writes. Thus, as with Convolution and Digit Recognition, we can save on-chip memory by disabling integrity counters. Since Affine Transformation accesses data at consistent chunks of 64B, we use 8 engine sets for inputs with a total 32KB buffer and 4 engine sets for outputs with a total 16KB buffer, all with one AES and HMAC engine. We observe overheads of 1.41x-2.22x.

DNNWeaver performs both streaming reads for weights and arbitrary accesses for feature maps. Weights are only read in large chunks, while feature maps require multiple reads and writes for small chunks. Since DNNWeaver's address space is difficult to partition, we provision two separate engine sets, but increase each set's engine parallelism. The weights engine set uses a large C_{mem} of 4KB, and 4 AES and 1 HMAC engine with total 128KB buffer and no integrity counters. The feature map engine set uses a smaller C_{mem} value of 64B, and similarly 4 AES and 1 HMAC engine with total 64KB of buffer. As the feature maps cover approximately 1MB of memory, 16KB of on-chip storage is used for integrity counters. In doing so, we achieve overheads of 3.20x-3.83x. These overheads are primarily due to DNNWeaver waiting for long HMAC computations for large 4KB chunks for weights before issuing more bursts. While the feature map engines also perform integrity checks, we efficiently cache these accesses by configuring a comparatively large buffer. To alleviate the bottleneck, we replace the HMAC module with 4 PMAC engines in the weight engine set, enabling higher authentication bandwidth for each memory transaction. This reduces the DNNWeaver overhead to 2.31x for the AES-128/16x case, down from 3.20x.

Bitcoin operates on small data (a 76 byte block header) and only outputs a 4 byte nonce. We optimize for area by simply leveraging the register interface, with one AES and one HMAC engine, to secure communication. Because Bitcoin performs significant computation for each input, we observe almost no overheads required to secure the accelerator.

Table 3 shows the percent resource utilization of the largest (e.g., AES/16x) Shield configuration across each accelerator. By matching the Shield configuration to the patterns and throughput of each accelerator, we provided a TEE with strong security at low performance and area overheads.

7 RELATED WORK

CPU Enclaves. Numerous flavors of secure processors and CPU enclaves exist [17, 29, 33, 34, 41, 42, 67, 74, 75, 85–87], with the best known being Intel SGX and ARM TrustZone. Keystone [64] is a recent framework for RISC-V enclaves that addresses CPU-specific memory-management challenges, such as self-paging and dynamic resizing, but does not provide hardware-enabled authenticated encryption. Keystone and other CPU-based enclaves rely on ISA extensions and hardware mechanisms not present in FPGAs, such as RISC-V PMP and SGX instructions. *ShEF* addresses orthogonal problems. Namely, *ShEF* is an end-to-end *framework* securing *arbitrary custom logic* in the face of challenges unique to cloud FPGAs. The flexibility of *ShEF* allows users to quickly leverage advances in secure processing, accelerate applications besides CPU enclaves, and quickly address the numerous vulnerabilities arising in secure processors [30, 61, 91].

Accelerator Enclaves. Graviton [94] modified the GPU peripheral hardware to protect against malicious device drivers from directly accessing sensitive resources, but treats DRAM as trusted memory. HIX [54] separated the driver out from the OS and ran it inside a trusted CPU enclave, essentially creating a heterogeneous trusted environment across both CPU and GPU, but requiring changes to the CPU and PCIe root complex. HETEE [105] proposed fabricating a tamper-resistant box of accelerators (namely GPUs) that a rack of servers can access via a centralized security controller in a dedicated FPGA. However, it did not consider the unique challenges that arise in cloud FPGAs (Section 2.4) and required a specialized tamper-resistant chassis. Telekine [51] addressed a novel side-channel in GPU enclaves in the context of ML training. In comparison, *ShEF* assumes a stronger threat model by neither trusting off-chip memory (including HBM) nor relying on a CPU enclave. *ShEF* requires no additional hardware or FPGA modifications. Border Control [73] sandboxed untrusted accelerators given access to system memory assuming a trusted CPU using similar techniques to *ShEF*, such as caching and auditing memory accesses. *ShEF* addresses the opposite problem of protecting accelerators from insecure system software and the Shell.

FPGA Security. FPGAs have been used to accelerate cryptographic primitives [38, 70]. There is also increasing interest in general secure computing on FPGAs. PFC [99] used proxy re-encryption to provide encrypted I/O to an accelerator, pre-programmed by the manufacturer, on cloud FPGAs. Cipherbase [25] accelerated encrypted database operations using an FPGA. Eguro and Venkatesan [36] used a trusted third party to sign and encrypt bitstreams to be loaded on remote FPGAs. CPU enclaves have been simulated in FPGAs [63, 64, 67]. Commercial FPGAs encrypt bitstreams via an embedded key, preventing adversaries from snooping on or modifying the user design [96], but requiring a trusted third party. Coughlin et al. extended this process by using self-provisioning keys in the FPGA hardware to eliminate the third party [35] and demonstrated a remote attestation protocol. MeetGo [72] and AMBASSY [52] discussed bootstrapping remote attestation to an embedded private

key and an ARM TrustZone processor, respectively. However, these works did not address key challenges described in Section 2.4, such as remote attestation within the cloud or the Shell, and required users to implement isolated execution and secure storage and I/O. Finally, Trimberger et al. [88] discussed security concerns of cloud FPGAs, including how to detect malicious user logic and prevent tampering with user logic. Mahmoud et al. [68] and Elnaggar et al. [39] presented attack methods in multi-tenant FPGAs.

8 CONCLUSION

As compute shifts towards cloud accelerators, the need for both secure and accelerated compute over sensitive data is dire. We address this need with *ShEF*, a framework consisting of a secure boot and configurable remote attestation process, as well as Shield logic that guarantees run-time isolated execution. We leverage FPGAs' reconfigurability to allow developers to craft a holistic and bespoke trusted execution environment (TEE) to fit their security and performance requirements. We prototyped *ShEF* on current FPGA hardware. We demonstrated secure boot and remote attestation on a local FPGA, enabled a secure storage application, and evaluated the Shield with several accelerators on AWS F1.

ACKNOWLEDGMENTS

We sincerely thank our shepherd, Dmitry Ponomarev, and the anonymous reviewers for their helpful feedback. We are also grateful to Xilinx for a generous equipment donation. This work was supported by the Stanford Platform Lab and its affiliates, as well as by a Stanford Graduate Fellowship.

A ARTIFACT APPENDIX

A.1 Abstract

In our artifact, we provide the entirety of the *ShEF* source code, including the Shield and implementations of the Secure Boot and Remote Attestation protocols. Our artifacts also include a number of reference benchmarks that we use to evaluate *ShEF* in Section 6. We provide instructions on how to build, run, and evaluate Shield benchmarks on AWS F1 instances. Our archival and GitHub repository also provides a README containing more details on using *ShEF*.

A.2 Artifact Checklist (Meta-Information)

- **Algorithm**: *ShEF* framework, including Shield, Secure Boot, and Remote Attestation protocols.
- **Program**: Includes custom vector-vector addition, matrix multiplication, SDP, and Bitcoin benchmarks. Also includes open-source benchmarks from Xilinx [1, 6], Rosetta suite [104], and DNNWeaver [81].
- **Compilation**: AWS F1 benchmarks compiled using AWS FPGA Developer AMI Version 1.8.1, Xilinx Vivado/SDAccel Version 2019.2, and Developer Kit Version 1.4.14. Development and compilation runs on a z1d.2xlarge EC2 instance.
- **Binary**: Source code for Ultra96 Firmware and Shield included, alongside scripts to generate FPGA bitstreams for AWS F1 instances.
- **Run-time environment**: AWS F1 bitstreams run on the AWS FPGA Developer AMI Version 1.8.1 with Developer Kit Version 1.4.14, running on an AWS EC2 f1.2xlarge instance.
- **Hardware**: Shield evaluations performed on an AWS EC2 f1.2xlarge instance.
- **Execution**: Shield evaluation performed using provided scripts.
- **Metrics**: Execution latency and resource utilization of various Shield configurations compared to baseline accelerator.
- **Output**: Execution latency reported by runtime binaries. Resource utilization reported by Xilinx Vivado compilation workflow.
- **Experiments**: Scripts and detailed workflow provided to compile and generate benchmarks (Figure 6)
- **How much disk space required (approximately)**?: About 10 GB for each compiled benchmark.
- **How much time is needed to prepare the workflow (approximately)**?: Around an hour to set up AWS F1 build environment.
- **How much time is needed to complete experiments (approximately)**?: Each benchmark bitstream compilation typically takes around 3-4 hours. Compiling all configurations for an accelerator can take a day, although different accelerators may be compiled in parallel using multiple terminals/VMs.
- **Publicly available**?: Yes
- **Code licenses (if publicly available)**?: ShEF is held under MIT. Portions (e.g., some source code and benchmark applications) may be held under different licenses, including BSD-2 and Apache License 2.0.
- **Archived (provide DOI)**?: Yes, at 10.5281/zenodo.5735634 and on GitHub at https://github.com/stanford-mast/ShEF.

A.3 Description

A.3.1 How to Access. The source code for the end-to-end ShEF workflow, as well as benchmarks, is archived at 10.5281/zenodo.5735634.

A.3.2 Hardware Dependencies. This AE requires access to AWS EC2 instances (z1d.2xlarge and f1.2xlarge).

A.3.3 Software Dependencies. Access to the AWS FPGA Developer AMI (https://aws.amazon.com/marketplace/pp/prodview-gimv3gqbpe57k?ref=cns_1clkPro) is required, which includes licenses for required Xilinx tools. Additionally, the AWS F1 Developer Kit is required and can be found on GitHub at https://github.com/aws/aws-fpga.

A.4 Installation

Obtain the code from the artifact repository or GitHub and follow the instructions in the README under *ShEF Shield Setup*. Specifically, follow instructions to setup an AWS account and permissions, the Developer instance using a *z1d.2xlarge* instance, and the Runtime instance using a *f1.2xlarge* instance.

A.5 Experiment Workflow

Specific benchmarks (namely DNNWeaver, bitcoin, and SDP) are located in `apps/<benchmark_name>`. Building and running each benchmark follows a similar workflow. We describe how to build and run `dnnweaver_shield`, a DNNWeaver plus Shield benchmark using LeNet, below.

First, build the bitstream using the Developer instance.

```
# Setup the env variable for the app
cd $SHEF_DIR/apps/dnnweaver_shield
export CL_DIR=$(pwd)
```

```
# Setup the build environment
source $AWS_FPGA_REPO_DIR/hdk_setup.sh

# Build the accelerator bitstream
cd $CL_DIR/build/scripts
./aws_build_dcp_from_cl.sh -foreground
```

The build script will take several hours (typically 2-3). Be sure to run in a `tmux` terminal or omit `-foreground` to use a nohup context so that SSH session disconnects do not terminate your build. Once the build is complete, submit your bitstream to AWS.

```
# Use the S3 bucket created during setup
aws s3 cp \
    $CL_DIR/build/checkpoints\
    /to_aws/*.Developer_CL.tar \
    s3://<bucket-name>/<dcp-folder-name>/

aws ec2 create-fpga-image \
    --region <region> \
    --input-storage-location \
    Bucket=<bucket-name>,\
    Key=<dcp-folder-name>/<tar-name> \
    --logs-storage-location \
    Bucket=<bucket-name>,\
    Key=<logs-folder-name>
```

Save the AFI and AGFI output by the command above. The AFI build will take around one hour. Run the following command and check that the status is `available` before proceeding.

```
aws ec2 describe-fpga-images \
    --fpga-image-ids <your-afi-id>
```

Finally, SSH into *Runtime* instance. Load the FPGA with the AFI, and build and run the test binary.

```
# Assuming same env variables as before
sudo su
source $AWS_FPGA_REPO_DIR/sdk_setup.sh

fpga-clear-local-image -S 0
fpga-load-local-image -S 0 -I \
    <your-agfi-id> # AGFI, not AFI
fpga-describe-local-image -S 0 -R -H

cd $CL_DIR/software
make
./test_lenet
```

The README also contains detailed instructions to build and run experiments.

A.6 Evaluation and Expected Results

Each benchmark consists of two root application directories, one representing the baseline and one with the Shield. For example, for DNNWeaver, the projects are at `apps/dnnweaver` and `apps/dnnweaver_shield`, respectively. Follow the instructions to build and run both as described above (the steps are the same for both applications). Compare the end-to-end latencies reported by both runtime binaries. For example, we observe a latency of 5073μs for `dnnweaver_shield` compared to 3054μs with `dnnweaver`. The resource utilization for a bitstream can be found in `$CL_DIR/build/reports/<timestamp>.SH_CL_all_utilization.rpt`, split up by module. `shield_wrapper_inst` shows the overall Shield resource utilization. Shield execution latency and resource requirements for a given benchmark are reported in Figure 6 and Table 3.

A.7 Experiment Customization

For each application, the Shield can be customized by modifying parameters included in `$SHEF_DIR/hdk/source/interfaces/free_common_defines.vh`. Please refer to the README for more information.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] [n. d.]. Affine Transformation. https://github.com/Xilinx/SDAccel_Examples/tree/master/vision/affine.
[2] [n. d.]. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1.
[3] [n. d.]. AWS EC2 FPGA Development Kit. https://github.com/aws/aws-fpga.
[4] [n. d.]. AWS Marketplace. https://aws.amazon.com/marketplace/search/results?x=0&y=0&searchTerms=fpga.
[5] [n. d.]. Capital One Data Breach Compromises Data of Over 100 Million. https://www.nytimes.com/2019/07/29/business/capital-one-data-breach-hacked.html.
[6] [n. d.]. CNN Convolution Example. https://github.com/Xilinx/SDAccel_Examples/tree/master/getting_started/clk_freq/large_loop_ocl.
[7] [n. d.]. Deneb Genetics. http://www.denebgenetics.com/.
[8] [n. d.]. Deploy a model as a web service on an FPGA with Azure Machine Learning service. https://docs.microsoft.com/en-us/azure/machine-learning/service/how-to-deploy-fpga-web-service.
[9] [n. d.]. Equifax Data Breach. https://epic.org/privacy/data-breach/equifax/.
[10] [n. d.]. F1 Acceleration for Montecarlo Financial Algorithms of FPGAs. https://www.xilinx.com/publications/events/developer-forum/2018-frankfurt/f1-acceleration-for-montecarlo-financial-algorithms-on-fpga.pdf.
[11] [n. d.]. FPGA-accelerated Cloud Server. https://www.huaweicloud.com/en-us/product/fcs.html.
[12] [n. d.]. FPGA-based AWS EC2 F1 instances for Cloud Network Security. https://valtix.com/blog/valtix_ec2_f1_sc19/.
[13] [n. d.]. FPGA Cloud Compute. https://cloud.baidu.com/product/fpga.html.
[14] [n. d.]. Huxelerate Standard Library. https://platform.huxelerate.it/docs/index.html.
[15] [n. d.]. Illumina DRAGEN on AWS. https://aws.amazon.com/quickstart/architecture/illumina-dragen/.
[16] [n. d.]. Intel Stratix 10 FPGA Applications. https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10/applications.html.
[17] [n. d.]. TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone.
[18] [n. d.]. Ultra96 Hardware User's Guide. http://zedboard.org/sites/default/files/documentations/Ultra96-HW-User-Guide-rev-1-0-V0_9_preliminary.pdf.
[19] [n. d.]. UltraScale+ FPGAs Product Tables and Product Selection Guide. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.
[20] [n. d.]. Xilinx Applications. https://www.xilinx.com/applications.html.
[21] [n. d.]. Xilinx Versal AI Core. https://www.xilinx.com/products/silicon-devices/acap/versal-ai-core.html#applications.
[22] 2016. Comparing the Speed of VM Creating 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/.

[23] 2017. Understanding and Profiling GCE Cold Boot Time. https://medium.com/google-cloud/understanding-and-profiling-gce-cold-boot-time-32c209fe86ab.

[24] 2020. *Intel Stratix 10 Device Security User Guide.* Technical Report. Intel Corporation.

[25] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure Database-as-a-Service with Cipherbase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1033–1036. https://doi.org/10.1145/2463676.2467797

[26] Daniel J Bernstein and Tanja Lange. 2017. Post-quantum cryptography. *Nature* 549, 7671 (2017), 188–194.

[27] Joppe W. Bos, Kristin Lauter, and Michael Naehrig. 2014. Private Predictive Analysis on Encrypted Medical Data. Cryptology ePrint Archive, Report 2014/336. https://eprint.iacr.org/2014/336.

[28] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[29] D. Champagne and R. B. Lee. 2010. Scalable architectural support for trusted software. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416657

[30] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *CoRR* abs/1802.09085 (2018). arXiv:1802.09085 http://arxiv.org/abs/1802.09085

[31] Jack Choquette and Wishwesh Gandhi. 2020. NVIDIA's A100 GPU: Performance and Innovation for GPU Computing. In *2020 IEEE Hot Chips 32 Symposium (HCS), Virtual, August 16-18, 2020*. IEEE.

[32] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[33] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086.

[34] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[35] Aimee Coughlin, Greg Cusack, Jack Wampler, Eric Keller, and Eric Wustrow. 2019. Breaking the Trust Dependence on Third Party Processes for Reconfigurable Secure Hardware. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. ACM, New York, NY, USA, 282–291. https://doi.org/10.1145/3289602.3293895

[36] Ken Eguro and Ramarathnam Venkatesan. 2012. FPGAs for Trusted Cloud Computing. In *International Conference on Field-Programmable Logic and Applications* (international conference on field-programmable logic and applications ed.). IEEE. https://www.microsoft.com/en-us/research/publication/fpgas-for-trusted-cloud-computing/

[37] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B. Lee, Nachiketh Potlapally, and Lionel Torres. 2009. *Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines.* Springer-Verlag, Berlin, Heidelberg, 1–22. https://doi.org/10.1007/978-3-642-01004-0_1

[38] A. J. Elbirt and C. Paar. 2000. An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher. In *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '00)*. ACM, New York, NY, USA, 33–40. https://doi.org/10.1145/329166.329176

[39] R. Elnaggar, R. Karri, and K. Chakrabarty. 2019. Multi-Tenant FPGA-based Reconfigurable Systems: Attacks and Defenses. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 7–12. https://doi.org/10.23919/DATE.2019.8714904

[40] Andre Esteva, Katherine Chou, Serena Yeung, Nikhil Naik, Ali Madani, Ali Mottaghi, Yun Liu, Eric Topol, Jeff Dean, and Richard Socher. 2021. Deep learning-enabled medical computer vision. *NPJ digital medicine* 4, 1 (2021), 1–9.

[41] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. 2014. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, United Kingdom) *(MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 190–202. https://doi.org/10.1109/MICRO.2014.25

[42] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. 2018. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1583–1600. https://doi.org/10.1145/3243734.3243743

[43] C. W. Fletcher, L. Ren, A. Kwon, M. v. Dijk, E. Stefanov, D. Serpanos, and S. Devadas. 2015. A Low-Latency, Low-Area Hardware Oblivious RAM Controller. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 215–222. https://doi.org/10.1109/FCCM.2015.58

[44] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. 2017. SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 357–380.

[45] Ilya Ganusov and Mahesh Iyer. 2020. Agilex Generation of Intel FPGAs. In *2020 IEEE Hot Chips 32 Symposium (HCS), Virtual, August 16-18, 2020*. IEEE.

[46] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme.* Ph. D. Dissertation. Stanford, CA, USA. Advisor(s) Boneh, Dan. AAI3382729.

[47] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Report 2016/204. https://eprint.iacr.org/2016/204.

[48] S. Hadjis and K. Olukotun. 2019. TensorFlow to Cloud FPGAs: Tradeoffs for Accelerating Deep Neural Networks. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 360–366. https://doi.org/10.1109/FPL.2019.00064

[49] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[50] M. Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 10–14. https://doi.org/10.1109/ISSCC.2014.6757323

[51] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. 2020. Telekine: Secure Computing with Cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 817–833. https://www.usenix.org/conference/nsdi20/presentation/hunt

[52] Dongil Hwang, Sanzhar Yeleuov, Jiwon Seo, Minu Chung, Hyungon Moon, and Yunheung Paek. 2021. Ambassy: A Runtime Framework to Delegate Trusted Applications in an ARM/FPGA Hybrid System. *IEEE Transactions on Mobile Computing* (2021), 1–1. https://doi.org/10.1109/TMC.2021.3086143

[53] Zsolt István, Soujanya Ponnapalli, and Vijay Chidambaram. 2021. Software-Defined Data Protection: Low Overhead Policy Compliance at the Storage Layer is within Reach! *Proc. VLDB Endow.* 14, 7 (March 2021), 1167–1174. https://doi.org/10.14778/3450980.3450986

[54] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 455–468. https://doi.org/10.1145/3297858.3304021

[55] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 1–12. https://doi.org/10.1145/3079856.3080246

[56] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 29–42. https://doi.org/10.1109/ISCA.2018.00014

[57] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with Amorphos. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 107–127.

[58] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[59] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. *SIGPLAN Not.* 53, 4 (June 2018), 296–311. https://doi.org/10.1145/3296979.3192379

[60] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 991–1010. https://www.usenix.org/conference/osdi20/presentation/roscoe

[61] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/woot18/presentation/koruyeh

[62] J. Krautter, D. R. E. Gnad, F. Schellenberg, A. Moradi, and M. B. Tahoori. 2019. Active Fences against Voltage-based Side Channels in Multi-Tenant FPGAs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942094

[63] I. Lebedev, K. Hogan, and S. Devadas. 2018. Invited Paper: Secure Boot and Remote Attestation in the Sanctum Processor. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 46–60. https://doi.org/10.1109/CSF.2018.00011

[64] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 38, 16 pages. https://doi.org/10.1145/3342195.3387532

[65] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[66] Vincent Liu, Mark A. Musen, and Timothy Chou. 2015. Data Breaches of Protected Health Information in the United States. *JAMA* 313, 14 (04 2015), 1471–1473. https://doi.org/10.1001/jama.2015.2252 arXiv:https://jamanetwork.com/journals/jama/articlepdf/2247135/jld150008.pdf

[67] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. ACM, New York, NY, USA, 311–324. https://doi.org/10.1145/2508859.2516692

[68] D. Mahmoud and M. Stojilović. 2019. Timing Violation Induced Faults in Multi-Tenant FPGAs. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1745–1750. https://doi.org/10.23919/DATE.2019.8715263

[69] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. 2020. Agile SoC Development with Open ESP. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD 20)*.

[70] Scott McMillan and Cameron Patterson. 2001. JBitsTM Implementations of the Advanced Encryption Standard (Rijndael). In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL '01)*. Springer-Verlag, London, UK, UK, 162–171. http://dl.acm.org/citation.cfm?id=647928.739896

[71] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*.

[72] Hyunyoung Oh, Kevin Nam, Seongil Jeon, Yeongpil Cho, and Yunheung Paek. 2021. MeetGo: A Trusted Execution Environment for Remote Applications on FPGA. *IEEE Access* 9 (2021), 51313–51324. https://doi.org/10.1109/ACCESS.2021.3069223

[73] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. 2015. Border Control: Sandboxing Accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 470–481. https://doi.org/10.1145/2830772.2830819

[74] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. 2013. OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2508859.2516678

[75] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh. 2020. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 776–789. https://doi.org/10.1109/ISCA45697.2020.00069

[76] Ed Peterson. 2018. *Developing Tamper-Resistant Designs with UltraScale and UltraScale+ FPGAs*. Technical Report. Xilinx, Inc.

[77] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC, USA, 183–196. https://doi.org/10.1109/MICRO.2007.44

[78] M. Sabt, M. Achemlal, and A. Bouabdallah. 2015. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. 57–64. https://doi.org/10.1109/Trustcom.2015.357

[79] F. Schellenberg, D. R. E. Gnad, A. Moradi, and M. B. Tahoori. 2018. An inside job: Remote power analysis attacks on FPGAs. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1111–1116. https://doi.org/10.23919/DATE.2018.8342177

[80] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.

[81] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[82] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh. 2019. FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review. *IEEE Access* 7 (2019), 7823–7859. https://doi.org/10.1109/ACCESS.2018.2890150

[83] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. 2019. MicroScope: Enabling Microarchitectural Replay Attacks. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 318–331. https://doi.org/10.1145/3307650.3322228

[84] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany) *(CCS '13)*. ACM, New York, NY, USA, 299–310. https://doi.org/10.1145/2508859.2516660

[85] G. E. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas. 2003. Efficient memory integrity verification and encryption for secure processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. 339–350. https://doi.org/10.1109/MICRO.2003.1253207

[86] Richard Ta-Min, Lionel Litty, and David Lie. 2006. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, Berkeley, CA, USA, 279–292. http://dl.acm.org/citation.cfm?id=1298455.1298482

[87] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS IX)*. ACM, New York, NY, USA, 168–177. https://doi.org/10.1145/378993.379237

[88] S. Trimberger and S. McNeil. 2017. Security of FPGAs in data centers. In *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*. 117–122. https://doi.org/10.1109/IVSW.2017.8031556

[89] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. Darwin: A Genomics Co-Processor Provides up to 15,000X Acceleration on Long Read Assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 199–213. https://doi.org/10.1145/3173162.3173193

[90] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1411–1428. https://www.usenix.org/conference/usenixsecurity19/presentation/von-gleissenthall

[91] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association.

[92] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[93] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[94] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 681–696. https://www.usenix.org/conference/osdi18/presentation/volos

[95] Martin Voogel, Yohan Frans, and Matt Ouellette. 2020. Xilinx Versal Premium Series. In *2020 IEEE Hot Chips 32 Symposium (HCS), Virtual, August 16-18, 2020*. IEEE.

[96] Kyle Wilkinson. 2018. *XAPP 1267: Using Encryption and Authentication to Secure an UltraScale/UltraScale+ FPGA Bitstream.* Technical Report. Xilinx, Inc.

[97] P. F. Wolfe, R. Patel, R. Munafo, M. Varia, and M. Herbordt. 2020. Secret Sharing MPC on FPGAs in the Datacenter. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 236–242. https://doi.org/10.1109/FPL50879.2020.00047

[98] L. Wu, D. Bruns-Smith, F. A. Nothaft, Q. Huang, S. Karandikar, J. Le, A. Lin, H. Mao, B. Sweeney, K. Asanović, D. A. Patterson, and A. D. Joseph. 2019. FPGA Accelerated INDEL Realignment in the Cloud. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 277–290. https://doi.org/10.1109/HPCA.2019.00044

[99] L. Xu, W. Shi, and T. Suh. 2014. PFC: Privacy Preserving FPGA Cloud - A Case Study of MapReduce. In *2014 IEEE 7th International Conference on Cloud Computing*. 280–287. https://doi.org/10.1109/CLOUD.2014.46

[100] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*. 640–656. https://doi.org/10.1109/SP.2015.45

[101] Chengliang Zhang, Junzhe Xia, Baichen Yang, Huancheng Puyang, Wei Wang, Ruichuan Chen, Istemi Ekin Akkus, Paarijaat Aditya, and Feng Yan. 2021. *Citadel: Protecting Data Privacy and Model Confidentiality for Collaborative Learning*. Association for Computing Machinery, New York, NY, USA, 546–561. https://doi.org/10.1145/3472883.3486998

[102] M. Zhao and G. E. Suh. 2018. FPGA-Based Remote Power Side-Channel Attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*. 229–244. https://doi.org/10.1109/SP.2018.00049

[103] Shijun Zhao, Qianying Zhang, Guangyao Hu, Yu Qin, and Dengguo Feng. 2014. Providing Root of Trust for ARM TrustZone Using On-Chip SRAM. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices* (Scottsdale, Arizona, USA) *(TrustED '14)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2666141.2666145

[104] 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. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. ACM, New York, NY, USA, 269–278. https://doi.org/10.1145/3174243.3174255

[105] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang, and D. Meng. 2020. Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1450–1465. https://doi.org/10.1109/SP40000.2020.00054