2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) | 978-1-6654-3555-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/FCCM51124.2021.00036

Trusted Configuration in Cloud FPGAs

Shaza Zeitouni*, Jo Vliegen†, Tommaso Frassetto*, Dirk Koch‡, Ahmad-Reza Sadeghi* and Nele Mentens†§
*Technische Universität Darmstadt, Germany, {shaza.zeitouni, tommaso.frassetto, ahmad.sadeghi}@trust.tu-darmstadt.de
†Katholieke Universiteit Leuven, Belgium, {jo.vliegen, nele.mentens}@kuleuven.be
‡The University of Manchester, UK, dirk.koch@manchester.ac.uk
§Leiden University, The Netherlands, n.mentens@liacs.leidenuniv.nl

***Abstract*—In this paper we tackle the open paradoxical challenge of FPGA-accelerated cloud computing: On one hand, clients aim to secure their Intellectual Property (IP) by encrypting their configuration bitstreams prior to uploading them to the cloud. On the other hand, cloud service providers disallow the use of encrypted bitstreams to mitigate rogue configurations from damaging or disabling the FPGA. Instead, cloud providers require a verifiable check on the hardware design that is intended to run on a cloud FPGA at the netlist-level before generating the bitstream and loading it onto the FPGA, therefore, contradicting the IP protection requirement of clients. Currently, there exist no practical solution that can adequately address this challenge.**

We present the first practical solution that, under reasonable trust assumptions, satisfies the IP protection requirement of the client and provides a bitstream sanity check to the cloud provider. Our proof-of-concept implementation uses existing tools and commodity hardware. It is based on a trusted FPGA shell that utilizes less than 1% of the FPGA resources on a Xilinx VCU118 evaluation board, and an Intel SGX machine running the design checks on the client bitstream.

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are popular implementation platforms for performance enhancement and energy saving in cloud computing. In a recent market search report, Grand View Research mentions that the increased adoption of FPGA resources in the cloud is an important driver for the growth of the FPGA market [1]. FPGAs are already used in commercial cloud platforms as in Amazon EC2 F1 [2], Microsoft Azure Catapult [3] and Alibaba Cloud F3 [4]. Examples of applications that benefit from FPGA acceleration in the cloud are artificial intelligence, financial trading and network encryption. The deployment model of cloud FPGAs is the following: when a client requests FPGA computing resources in the cloud, the cloud service provider (CSP) allocates an FPGA instance (from a pool of FPGA instances) to the client for a specific amount of time.

Recent research presents attacks specific to the FPGA hardware on commercially deployed cloud FPGAs. These attacks allow clients to potentially damage FPGAs hosted in the cloud and consequently disable the computing resources of other clients. A malicious client performs such a denial-of-service (DoS) attack by uploading a design circuit that drains an excessive amount of current from the power supply of the FPGA such that the whole platform stops functioning. This can be done, e.g., with ring oscillators [5], [6] or by invoking short circuits [7]. A real-world DoS attack on EC2 F1 instances is demonstrated in [8].

Other categories of attacks – side and covert-channel attacks – have been mostly shown in academic settings, assuming that different clients share different portions of the same FPGA. This allows a malicious client to exploit crosstalk between the wires of an FPGA [9], [10], voltage fluctuations on the shared power distribution network [11]–[13] or thermal information [14]. For example, to perform a side-channel attack, a malicious client uploads a sensor circuit onto the allocated portion of the FPGA with the goal to retrieve secret data processed by another client sharing the same FPGA fabric simultaneously. While no CSP offers the concurrent use of a single FPGA device among clients, multiple computing resources, including FPGAs, typically share a power supply rail, which represents an attack surface [15], [16]. A detailed analysis of these attacks and their defenses is presented in [17].

Therefore, it is important for CSPs to detect the presence of circuits that have sensor or power draining capabilities before they are loaded on the FPGA. We refer to these circuits as *rogue circuits* in the remainder of this paper. The only way to prevent a rogue circuit from being configured onto an FPGA, is to perform a check on the circuit the client intends to upload. In existing commercial platforms, this check is done by the CSP using the FPGA vendor tools that inspect the netlist, e.g., running Xilinx design rule checks (DRCs). Initiatives in academic research propose the use of virus scanners to check FPGA bitstreams [18], [19].

However, any detection mechanism, be it through the existing commercial platforms or through academic virus scanner tools, needs access to the configuration data representing the circuit that the client intends to upload. Consequently, the client is forced to reveal the Intellectual Property (IP) of the hardware circuit to the CSP, which may violate IP protection policy for companies. Clients would rather send encrypted configuration bitstreams to the service provider. However, the use of encrypted bitstreams does not comply with the requirement of the CSP to check the incoming FPGA configurations before they are loaded onto the FPGA.

Contributions. In this work, we propose `TruFPGA`, the first scheme to satisfy the requirements of both, the clients in protecting their IPs and the CSP in guaranteeing that a check has been done on the presence of rogue circuits in the configuration bitstream. In `TruFPGA`, a design check on a client bitstream is executed in a Trusted Execution Environment (TEE) and a proof of execution is provided to the CSP. We further present two options where the TEE resides either on the client side or CSP side and show the trade-offs between the two options. The CSP loads the encrypted bitstream on the FPGA, where it is decrypted on-the-fly with the help of a trusted FPGA shell that we designed. As such, the CSP has no access to the decrypted bitstream, thus solving the paradox of client's IP protection while preventing rogue FPGA configurations. In `TruFPGA`, we tackle the following challenges:

(1) Secret session keys need to be securely established between cloud FPGAs and clients. This is non-trivial in cloud FPGAs because (a) multiple clients with different secret keys should be supported over time, (b) clients are oblivious to the identities of the FPGA instances assigned to them, i.e., CSP's proprietary information, and (c) the CSP, who controls and configures the FPGA, should not have access to the secret keys used in the FPGA.

(2) The client needs the assurance that (a) a read-out of the FPGA's configuration memory, including client's IP, is disallowed, and (b) an unauthorized configuration of client's application bitstream is denied. This is technically challenging, because the CSP controls the FPGA.

(3) The overall solution must be efficient and incurs minimal or no changes to (a) the FPGA architecture, and (b) the cloud infrastructure, e.g., avoid direct communication channel between clients and cloud FPGAs.

We tackle these challenges in `TruFPGA` that comprises a *trusted shell* on the FPGA leveraging physically unclonable functions (PUFs) for key generation and an overarching security *protocol* between the involved parties. Our *proof-of-concept* implementation is demonstrated with existing tools and commodity hardware. Further, `TruFPGA` is generic and the necessary design checks can be performed through vendor toolchains or academic virus scanner tools.

II. Background

In this section we present a brief background on the security components and concepts used in `TruFPGA`.

A. Trusted Execution Environment (TEE)

Ideally, a TEE guarantees that the code and the data running inside the TEE are protected with respect to confidentiality and integrity. Commercial TEEs include Intel SGX [20], AMD SEV [21] and ARM TrustZone [22]. Being a subject of an active research field, several TEE architectures have been proposed, e.g., Sanctum [23], Sanctuary [24], Keystone [25] and Cure [26]. A TEE is an execution environment with its own hardware and software components. Typically, in a TEE, a security-sensitive application, referred to as an *enclave*, runs in isolation of all software on the system including the untrusted operating system (OS) or the hypervisor. A host process, e.g., the OS, sets up the enclave. This means that the enclave's initial binaries may be manipulated. Therefore, the authenticity and integrity of the enclave's initial binaries are verified before execution through remote or local *attestation*. Only then, confidential data can be communicated to the enclave over a secure channel.

B. Remote Attestation

It enables a party/entity to verify the authenticity and integrity of a piece of code or memory on a remote device. A trust anchor on the device computes a digest of the code or memory content, e.g., using a cryptographic hash function and a secret key shared with the verifier. The verifier compares the received digest to a reference value to verify the remote device's status. In the case of a TEE, the platform's secret key is used for attestation.

C. Physically Unclonable Function (PUF)

Silicon PUFs leverage the uncontrollable manufacturing process variation of integrated circuits as a source of entropy to derive a device-specific cryptographic key or a unique identifier. A PUF is stimulated by an input, *challenge*, to produce a *response*, which depends on both the challenge and the innate physical characteristics of the PUF circuit. Therefore, PUF responses are envisioned to be unique and unpredictable. Nevertheless, PUFs have been shown to be prone to software-based modeling attacks [27]. Such attacks require the collection of a large number of challenge-response pairs (CRPs) of a PUF instance to build a mathematical model that emulates the intended PUF behavior. One of the solutions to mitigate such attacks is to obfuscate the output of a PUF using, for example, a cryptographic hash function, such that an attacker has no access to the actual CRPs of a PUF. Note that PUF-based secret keys can be generated on-the-fly, thus eliminating the need for secure non-volatile key storage. PUF technology has been widely adopted for digital fingerprinting and authentication of IoT devices. Moreover, PUFs have already made their way into some FPGA families, e.g., Intel Stratix-10, and Microsemi SmartFusion-2 for the generation of device-specific secret keys.

III. System and Trust Model

In a typical cloud-computing paradigm, different parties are involved. Cloud service providers *CSPs*, such as Microsoft or Amazon, provide different usage models and services and deploy heterogeneous computing platforms. Such platforms involve a conventional CPU-based host that interfaces with co-processors and accelerators (GPUs, ASICs, FPGAs, etc.), which are in turn supplied by different hardware vendors. FPGAs and FPGA design tools are provided by *FPGA vendors*. The CSP may deploy FPGAs of one or more vendors. Computation capacity is rented by a *client* that communicates the workload, i.e., code and data, to the CSP. Workloads of multiple clients may share the same physical resources in the cloud, according to the allocation and scheduling policies of the CSP. The current FPGA deployment model of commercial CSPs, which is also adopted in this work, allocates an entire FPGA instance from an FPGA pool in the cloud to one client for an agreed amount of time. The protection mechanism we propose, enters into force when the CSP allocates a specific FPGA to the client.

A. Trust Model & Assumptions

Trust relations among the involved parties are as follows.

FPGA vendor: the CSP as well as the clients trust the FPGAs and the design tools offered by the FPGA vendor.

Client-Client: co-clients sharing physical resources in the cloud are mutually distrusting.

CSP-Client: the CSP does not trust clients in general. More specifically, a malicious client may launch physical attacks remotely through rogue FPGA configurations. In such attacks, a malicious client implants virus circuits in the design intended to run on the cloud FPGA to mount various remote physical attacks, e.g., DoS attacks that could lead to shutting down cloud services.

Client-CSP: clients do not trust the CSP with their sensitive data. The CSP owns the infrastructure and is motivated by reputation and financial gain, therefore, DoS and physical attacks on the cloud infrastructure by the CSP are excluded.

Assumptions. We focus in this work on cloud FPGA configurations that intend to shut down FPGAs or perform side/covert-channel attacks using rogue primitive circuits. As such, attacks that do not deploy rogue primitives are not considered [28]. We assume that FPGA vendors are willing to support IP protection on cloud FPGAs. This is consistent with the assumptions in related work [29]–[31] and is acceptable by FPGA vendors, e.g., this is evident in Intel Stratix-10 FPGAs where Intel supports remote secure key provisioning [32]. Further, we assume the CSP, the clients, and the FPGA vendors communicate with each other over secure channels, e.g., TLS, to prevent man-in-the-middle attacks. We assume the CSP offers the clients, upon request, to run their security-sensitive applications in TEEs. Finally, we assume that standard cloud security measures and protection of software against various attacks are in place.

B. Objectives & Requirements Analysis

We summarize the objectives of this work and extract the technical requirements that allow our design in § IV to achieve the objectives under the trust relations among different parties.

Objectives. While the clients aim to protect their IP designs in cloud FPGAs, CSPs are keen to protect their infrastructure, including their FPGAs, against damage caused by rogue configurations in line with recent findings [5], [6], [8], [14]–[16].

Requirements. Based on the aforementioned objectives, we derive the following requirements.

R1: TEEs on cloud FPGAs. To protect confidentiality and integrity of the client's workload on a CPU in the cloud, CSPs increasingly offer the option to run the client's workload in a TEE (see § II), both in bare-metal instances [33], [34] and in virtual machines [35]. Analogously, such protection should be offered for workloads intended to run on cloud FPGAs. For instance, a client, who intends to run a machine learning (ML) model (trained on the client's private data) on a cloud FPGA, may want to protect the ML model against attacks that aim to extract the client's private data [36]. TEEs on cloud FPGAs can be used to achieve such protection. We refer to the components that establish the TEE on an FPGA as the *trusted shell*. Ideally, the trusted shell should be i) realized with the configurable fabric to allow for future patches, while ii) incurring minimal or no changes to FPGA architectures. This implies that protecting the integrity of the configurable trusted shell against unauthorized changes is a requirement to protect the TEE on the FPGA. Therefore, establishing TEEs on cloud FPGAs requires FPGA vendor support. This is akin to hardware vendor support to establish TEEs on CPUs, e.g., Intel SGX, and is justified, since hardware and FPGA vendors are implicitly trusted by other parties.

R2: Verifiable proof of virus-free FPGA bitstreams. One typical approach to provide the CSP with adequate assurance against known FPGA attacks is to check the client's FPGA design using proprietary tools or virus scanners [18], [19] that search for known virus signatures. For example, Amazon AWS runs vendor DRCs on the client's netlist to prevent combinatorial ring oscillatorsbefore generating the final bitstream. However, we assume that the client's bitsteam is encrypted, and hence the CSP has no access to the client's netlist or bitstream in plain. Therefore, a convincing proof must be provided to the CSP that the encrypted bitstream will pose no threat to the infrastructure as well as co-tenants.

IV. TruFPGA

To achieve the objectives in § III-B, we propose the *TruFPGA protocol*, which leverages TEEs on both CPUs and FPGAs. We further design a *trusted shell* that protects client's IP bitstream on cloud FPGAs and propose to run the design check inside the TEE, whether on the client side or the CSP side. Thus, an authentic report on the status of client bitstream can be generated inside the TEE enclave and provided to the CSP. Thus, fulfilling both requirements *R1* & *R2* (§ III-B). TruFPGA protocol, depicted in Fig. 1, consists of three phases: preparation (not shown in Fig. 1), offline (2 steps) and online (5 steps) phases.

Offline phase. In step ①, the FPGA vendor provides the client with the trusted shell, which will be installed on the cloud FPGA in the online phase, as well as with the key material that enables the protection of client bitstream. In step ②, client's encrypted bitstream is sent to the CSP to be checked for rogue circuits inside a TEE. By the end of this phase, the CSP either approves the client bitstream and proceeds with the online phase or aborts if the bitstream does not pass the virus check.

We opt for the TEE at the CSP side for the following reasons: i) CSPs are assumed to continuously maintain their infrastructures and deploy suitable defenses against known attacks, including attacks on TEEs (as discussed in § III-A), and ii) powerful computation resources on the cloud enable access to a TEE equipped with the required memory resources for the virus scanner. Nevertheless, the TEE can also be on the client side as we discuss next in § VII.

Online phase. In step ③, the CSP configures the trusted shell on the FPGA. As discussed in § III-B, the main objective of the trusted shell is to establish a TEE on cloud FPGAs. In

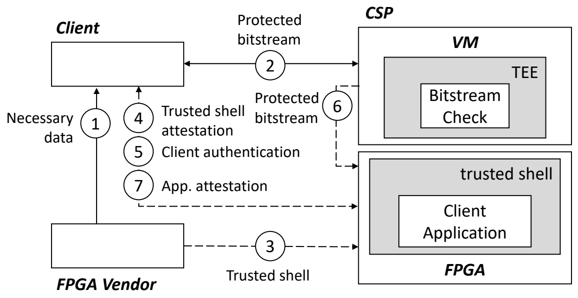

Fig. 1. TruFPGA: high-level overview. The straight arrows refer to the offline phase and the dashed arrows refer to the online phase of the protocol.

step ④, the client attests, i.e., verifies the integrity of, the trusted shell on the FPGA to ensure that the trusted shell has not been manipulated after configuration on the FPGA and that configuration readback is deactivated for the CSP. In step ⑤, the client authenticates itself to the FPGA to prevent unauthorized configuration of client bitstream (more details in § V). While in step ⑥, the CSP forwards the encrypted bitstream received in step ② to the intended FPGA for partial reconfiguration. Finally, in step ⑦, the client attests the intended application is configured on the intended FPGA. By the end of this phase, the client's application is ready to run on the cloud FPGA.

Next, we present our trusted shell in § IV-A and the detailed computations and communication steps of TruFPGA protocol in § IV-B.

A. Trusted Shell

1) Tasks: After its configuration on a cloud FPGA, the trusted shell exchanges data and receives the encrypted bitstream (for partial reconfiguration) from the CSP through a PCIe interface. The trusted shell's security-related tasks are:

Preventing read-out. The trusted shell blocks configuration memory read-out to protect the client's application. Only the trusted shell itself will have access to the FPGA configuration through the internal configuration access port.

Attestation of the trusted shell. The entire configuration memory of the FPGA, including the trusted shell, is read to compute a proof of integrity PoI_F using a nonce N_i (a random number used once in a cryptographic protocol) sent by the client and the secret key R_i. The attestation of the trusted shell is inspired by the FPGA self-attestation in [37].

Client authentication. The client must prove to the trusted shell that it knows the secret key R_{i+1} used for the encryption of the client bitstream. The client computes the proof of authenticity PoA_C over the FPGA configuration using a nonce N_{i+1} that is generated by the trusted shell and communicated to the client, and the secret key R_{i+1}. To verify PoA_C, the trusted shell also computes the proof of authenticity PoA_F and compares it to the client's PoA_C. Only upon successful authentication, secret key R_{i+1} is provided to the next task. Details on the computation of authenticity and integrity proofs are presented in § IV-B.

Bitstream verification & decryption. Only when the client authentication succeeds, i.e., when $PoA_C = PoA_F$, decryption and configuration of the client bitstream on the FPGA is permitted. The FPGA verifies the application bitstream integrity and decrypts it using the secret key R_{i+1} to obtain and configure the plain partial bitstream on the FPGA.

Some of these tasks require secret keys. We consider PUF-based secret key generation, as this approach binds the cryptographic keys to a specific FPGA instance and eliminates the need for permanent secret key storage, i.e., keys are generated on-the-fly, thus, minimizing the physical attack window. PUF-based secret key generation has been thoroughly investigated for FPGAs [38]–[41]. To prevent PUF modeling attacks [27], we use a controlled PUF (*CPUF*). The FPGA vendor enrolls the CPUF on each cloud FPGA before deployment and possesses a database of its CRPs for later use. PUF enrollment can be also performed by another trusted 3rd party.

Note that the trusted shell can be designed to accept plain partial bitstreams for clients that require no IP protection. Detailed architecture of the trusted shell is presented in § VI.

2) Design Space: Ideally, the trusted shell should be implemented in configurable logic. The advantage is two-fold; no further changes to commodity FPGAs are required and security or functional patches are feasible. This is vital, since changes to the trusted shell might be required to patch security bugs in cryptographic cores, e.g., the bitstream decryption core, as the recent work of Ender et al. [42] demonstrated an attack against the unpatchable decryption core on Xilinx 7-Series FPGAs. On the other hand, this implies protecting the integrity of the trusted shell itself prior to configuration, since a malicious party can manipulate the trusted shell to leak the secret keys. Therefore, we explore the trade-offs and propose two approaches for the trusted shell implementation, both of which fulfill requirement *R1* from § III-B and require the support of the FPGA vendor:

Fully-configurable trusted shell. In this approach, the entire trusted shell is implemented in configurable logic. To protect its integrity prior to configuration on cloud FPGAs, we rely on existing hardened authentication cores on commodity FPGAs, e.g., RSA-based authentication on Xilinx UltraScale FPGAs. The FPGA vendor enforces bitstream authentication on the FPGA, programs the public key on the FPGA before shipping it to the CSP and signs the trusted shell bitstream with the corresponding private key. This prevents the FPGA from loading an unauthorized trusted shell bitstream. We implement this approach in commodity FPGAs in § VI.

Hardening some components of the trusted shell. As an alternative approach, we propose to harden the components that perform remote attestation and client authentication tasks. These are the CPUF, the cryptographic core that computes the authenticity and integrity proofs and the finite state machine (FSM) that controls them. This further requires to harden the bus carrying the configuration data from the configuration engine to the cryptographic core to ensure that the proofs are computed on the actual configuration data. Note that the computation of the proofs can be also performed by a hardened processor/microcontroller instead of the cryptographic core and the FSM, given that the firmware implementing these instructions is protected by the FPGA vendor. The hardened components cannot be modified or altered by malicious parties, thus, they form a root of trust in the FPGA. By leveraging remote attestation, the trust is extended to the other configurable components of the trusted shell, i.e., the verification & decryption core and the configuration memory controller that controls the configuration engine through the internal interface. Note that in recent FPGA families, e.g., Intel Stratix-10 and Microsemi SmartFusion-2, hardened PUF technology is already deployed for the generation of device-specific secret keys.

B. `TruFPGA`: Protocol

This section explains the detailed computations and exchanged messages in the `TruFPGA` protocol that is presented at a higher level in Fig. 1 and detailed in Fig. 2. `TruFPGA` requires no direct communication channel between the client and the FPGA: the client uses the established secure channel with the CSP, who has full control over the FPGA, to communicate with the FPGA. Thus, the CSP has access to all messages sent to/from the FPGA.

1) Preparation Phase: **Step ⓪: PUF enrollment.** Prior to deployment in the cloud, the FPGA vendor enrolls the CPUF on each FPGA instance and collects a large number of CRPs. The CRPs are securely stored in a database at the FPGA vendor side.

2) Offline Phase: **Step ①: Acquire necessary data.** To rent a cloud FPGA, the client sends a *service request* to the CSP. The CSP then assigns an FPGA to the client according to the CSP allocation and scheduling policies. The CSP sends an obfuscated identifier of the allocated FPGA, $FPGA_{ID}$, to the client. Since the infrastructure information of the CSP is proprietary, the CSP and FPGA vendor can share a list of pseudo identifiers (PIDs) for each FPGA instance, such that a PID is never given twice. Alternatively, the actual FPGA identifier is encrypted using a secret key between the CSP and FPGA vendor, such that the client only sees an encrypted message.

The client forwards the $FPGA_{ID}$ to the FPGA vendor and gets back two messages. The first message contains the trusted shell Tsh, a nonce N_i, a challenge C_i, and a reference proof of integrity of the trusted shell PoI_V. The FPGA vendor computes the proof of integrity as follows: $PoI_V = HMAC(R_i, N_i||CD)$. Such that, HMAC is a keyed-hash message authentication code used to verify both the integrity and the authenticity of data, R_i is the secret key used in the HMAC and corresponds to the response of the CPUF to the challenge C_i: $R_i = CPUF(C_i)$, and CD is the configuration data of the targeted FPGA. The second message from the FPGA vendor to the client contains the CRP (C_{i+1}, R_{i+1}).

Step ②: Bitstream check. The client encrypts the application partial bitstream pBS_A using R_{i+1} and sends it to the CSP. An enclave on a TEE residing on the CSP side

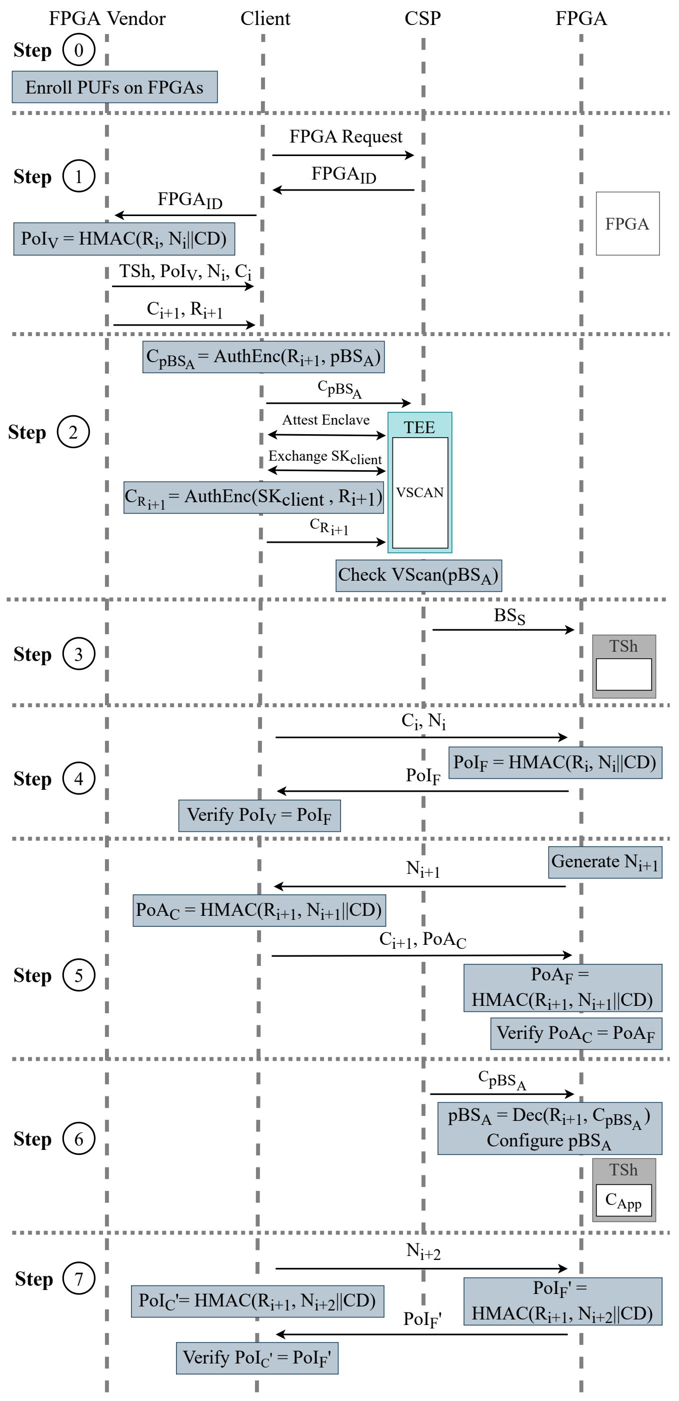

Fig. 2. `TruFPGA` Protocol. $AuthEnc()$: authenticated encryption algorithm, $Dec()$: decrypt and verify algorithm, and $VScan()$: virus scanner algorithm.

is initiated to check the application bitstream against known virus signatures. The enclave code includes a key exchange algorithm, a decryption algorithm and the virus scanner code. The client first attests the enclave binaries [43]. After enclave attestation, the client exchanges a session key SK_{client} to establish a secure link with the enclave. Through this secure channel, the secret key R_{i+1} is sent to the enclave to decrypt the application bitstream prior to the virus scan. The CSP receives only a report generated by the virus scanner about the bitstream status. The CSP has neither access to the secret key R_{i+1} nor to the plain application bitstream. The protocol proceeds if the bitstream is proven to be virus-free.

3) Online Phase: In this phase, the protocol is aborted, if any of the following steps fails.

Step ③: Trusted shell configuration. We assume the trusted shell is an open-source IP core provided by the FPGA vendor. The CSP configures the trusted shell bitstream BS_S on the intended FPGA. The trusted shell next deactivates all external configuration ports. Note that the trusted shell can be also provided by a 3rd party, therefore, it must be checked for rogue circuits before configuration on the FPGA.

Step ④: Trusted shell attestation. In this step the client verifies the installation of the trusted shell on the FPGA. Therefore, the client sends the challenge C_i and the nonce N_i, which are received from the FPGA vendor, to the FPGA. The challenge C_i is fed to the CPUF and the resulting response R_i is used to compute the integrity proof on the entire configuration memory, which contains at the moment the trusted shell only: $PoI_F = HMAC(R_i, N_i||CD)$. The CSP sends PoI_F back to the client for comparison to the reference attestation report PoI_V. In this context, remote attestation provides not only a means to verify the integrity of the trusted shell and the deactivation of external configuration ports, but also a proof of execution on the intended FPGA.

Step ⑤: Client authentication. Next, the client authenticates itself to the FPGA. For that, the trusted shell generates a fresh nonce N_{i+1} and sends it to the client. The client then computes $PoA_C = HMAC(R_{i+1}, N_{i+1}||CD)$ and sends it back to the trusted shell, together with C_{i+1}. The trusted shell computes $PoA_F = HMAC(R_{i+1}, N_{i+1}||CD)$, using $R_{i+1} = CPUF(C_{i+1})$, and verifies that $PoA_C = PoA_F$.

Step ⑥: Application bitstream configuration. The CSP sends the encrypted application bitstream to the intended FPGA. The trusted shell decrypts C_{pBS_A} into the plain bitstream pBS_A using the secret key R_{i+1}. The trusted shell then configures the application bitstream on the FPGA. As a result, the client application, denoted as C_{App}, is loaded on the FPGA and ready for use. Note that we use the same secret response R_{i+1} for client authentication and bitstream decryption to associate the bitstream to the client. Therefore, it is not possible for the CSP or other entities to load a different encrypted application bitstream after client authentication.

Step ⑦: Application attestation. As in step ③ the client attests the *entire* configuration memory of the FPGA including the application to ensure the intended application is running on the intended FPGA.

V. Security Analysis

In this section, we discuss possible attacks on `TruFPGA`. We assume that the cryptographic cores, i.e., the decryption core and the HMAC core, are cryptographically secure.

Pirated shell. To prevent the configuration of a pirated shell, whose target is to leak PUF-based keys or to compromise the computation of authenticity or integrity proofs. We discuss in § IV-A, design space, two approaches to prevent a pirated shell and to protect the integrity of the trusted shell: i) enforcing authentication on the cloud FPGA, thus only an authentic shell bitstream is configured or ii) hardening some of the components of the trusted shell.

Unauthorized configuration of client's IP. A CSP has access to all exchanged messages between the user and the allocated FPGA. After the client releases the FPGA, the CSP may use the same FPGA and instantiates it with the trusted shell. The CSP then replays prior exchanged messages with the trusted shell in order to run the client's application on the FPGA. In order to prevent unauthorized configuration of the client encrypted bitstream on the same FPGA, we add client authentication (step ⑤). In this step, the client or the party communicating with the FPGA must prove the possession of the secret response R_{i+1} used for bitstream encryption. The fresh nonce N_{i+1}, which is used to compute the proof of authenticity PoA_C, must be therefore *truly random* and is generated by the trusted shell. Note that a pseudo-random number generator (PRNG) cannot be used, because PRNG will generate the same nonce each time the trusted shell is configured on the FPGA (thus the CSP can replay recorded PoA_C). Therefore, the CSP cannot compute PoA_C since the CSP has no access to the secret R_{i+1}.

TEE security. Secure TEEs are expected to provide three properties: integrity, confidentiality, and secure remote attestation. These properties are essential to `TruFPGA` in order to ensure the integrity of the enclave code and the confidentiality of the bitstream (or the integrity of the report generated by the virus scanner in the case the TEE is assumed running on the client side, see § VII). Recent attacks on TEEs focused on extracting secret information, e.g., extracting attestation keys to spoof attestation reports [44], [45] or introducing small changes in TEE execution [46], [47]. However, influencing the computation inside a TEE in a meaningful way has not been shown yet. In response, TEE vendors are continuously patching their hardware vulnerabilities [48]. Nevertheless, TEE security is an orthogonal problem to the problem we are tackling in this paper. TEE security is an active field of research and new TEE architectures [24]–[26] are developed that try to tackle the weaknesses of current TEEs. Moreover, `TruFPGA` is TEE-agnostic and can seamlessly migrate to newer TEE technologies.

Configuration memory read-out. Major FPGA vendors allow reading back the FPGA's configuration memory after bitstream configuration and also provide the clients with the option to block this feature to prevent unauthorized read-out. However, in the cloud setting, the client has no physical access to the FPGA to enforce/ensure that this feature is blocked. Therefore, the trusted shell is designed to block configuration memory read-out after it is configured on the FPGA. Once the trusted shell is configured, the CSP cannot read out the client's application in plain form.

PUF security. As discussed in § IV-A PUF enrollment is assumed to be done by the FPGA vendor in a secure facility before shipping the FPGAs to the CSP. To prevent PUF

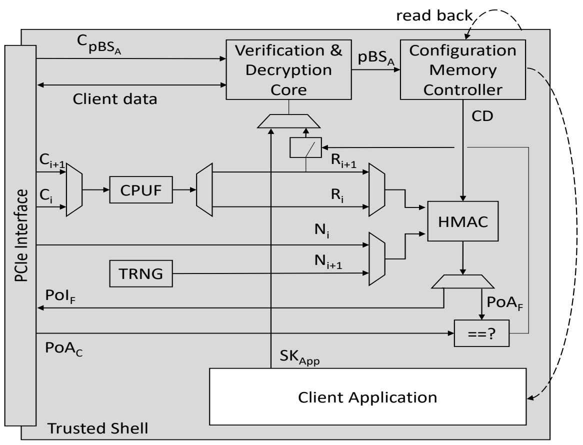

Fig. 3. Trusted shell architecture.

emulation or modeling attacks [27], we deploy a controlled PUF (CPUF) [49], [50]. As such, actual raw responses are processed internally, within the CPUF circuit, to generate the final responses. This prevents a malicious party from collecting raw CRPs for modeling attacks. Further, the FPGA vendor can set a quota of CRPs per client/day targeting the same FPGA.

VI. TruFPGA: Implementation & Evaluation

A. Trusted Shell

We prototype a configurable trusted shell, depicted in Fig. 3, on a Xilinx VCU118 evaluation board.

Components. We describe each of the components and list their required resources. The configuration memory controller is implemented using the AXI HWICAP core, which interfaces with the internal configuration access port (ICAP) primitive. The main tasks of the configuration memory controller (1161 LUTs, 1451 FFs, a BRAM, an ICAPE3) are to i) prevent configuration memory read-out ii) read the configuration memory for the computation of PoI_F & PoA_F, and iii) partially configure the application bitstream. Readback of configuration memory is disabled by setting the security bits SBITS in the Control Register to $1x$, which disables writes and reads through external configuration ports, but not to the ICAPE3 [51]. The verification & decryption core (3171 LUTs and 1004 FFs) includes an AES-128 core for bitstream decryption in the counter mode as well as a keyed hash function (AES-CMAC) for the integrity verification of the incoming client application bitstream. Note that for higher security assurance, a side-channel resilient AES core with higher security parameter (key width of 256-bit) can be used. The HMAC (1955 LUTs and 1455 FFs) for the computation of the integrity/authenticity proofs is implemented also with an AES-CMAC core with the 128-bit secret key generated by a CPUF circuit. In our prototype, we implemented a TRNG core for fresh nonces (1069 LUTs and 137 FFs) and a dummy CPUF that generates a random value for demonstration purposes only. However, we propose the use of CPUF as in [49], [50]. Commercial soft PUF IP cores are also available, e.g., Intrisic-ID Inc. (https://www.intrinsic-id.com). Note that the trusted shell must be designed to keep no records of secret keys after their usage to minimize the physical attack window. The trusted shell clears all intermediate key-related values immediately after their usage. Further, the trusted shell can be designed to configure unencrypted IP bitstreams directly for clients that do not need IP protection.

Integrity Protection. We leverage RSA authentication, which can be used independently of bitstream encryption [51], to authenticate the static trusted shell bitstream before configuration. RSA authentication is enforced by setting the OTP eFUSE Security Register (FUSE_SEC) [51], thus, only authentic bitstreams can be configured. FPGA vendor support is required to program the RSA public key and enforce the authentication prior to deployment in the cloud.

B. Virus Scanner in the TEE

In order to prove the feasibility of performing privacy-preserving checks on clients' bitstreams, we run the virus scanner on a commodity TEE with limited memory resources resembling a client machine for demonstration purposes only. We run our experiments on a computer with modest resources (Core i7-7700 CPU clocked at 3.6 GHz and 8 GB of RAM) running Ubuntu 18.04.4 LTS. We use the Graphene-SGX framework [52] to embed the virus scanner into an Intel SGX enclave [20]. We chose the open-source virus scanner FPGADefender [53], which is designed to detect short-circuits and self-oscillating circuits in bitstreams. We test our setup with a number of FPGA designs used also in [53] and implemented on a on a Zynq UltraScale+ MPSoC: parallel scrambler, stepper motor, encoder/decoder, coded decimal adder, RS_232 UART, I2C Bus, DES, AES, SHA3, PSNG, TRNG, SPI, CAN controller, Cordic, MIPS CPU, RISC-V CPU, Mandelbrot and FPGA miner. Due to the limited space, we report the runtime of the biggest tested design, SHA3 core, which is approximately 36 minutes. The runtime overhead of SHA3 compared to the runtime of unprotected version is $8.82\times$, whereas the geometric mean of the runtime overheads of the aforementioned designs is approximately $3.21\times$. This is due in our setup SGX can only access up to 128 MB of encrypted memory at a time. Therefore, for benchmarks that require more than 128 MB, the SGX driver for Linux relies on paging, i.e., least-recently-used encrypted pages are replaced when other pages are needed. Note that optimizing the virus scanner performance, as evident in [8], will significantly improve our results. To account for bigger FPGAs, e.g., with multiple super logic regions (SLR), the client can provide a bitstream for each SLR to be scanned individually. The CSP then approves the client design when bitstreams of all SLRs are scanned. To further reduce the overhead, the scan can be done once per client's bitstream. This is feasible by keeping encrypted and integrity-protected records, e.g., the scan report and a hash of the bitstream, of scanned bitstreams. The enclave then checks whether the incoming bitstream is within the list of records, by comparing its hash to existing hash values. The bitstream gets scanned, only if it is not scanned before.

VII. Discussion

Location of the virus scanner. It is also feasible to scan the bitstream in a TEE on the client side. However, it requires slight modifications to step ②. The enclave code includes the virus scanner, an encryption algorithm and an algorithm to compute $PoI_{Enclave}$ of both, the resulting scan report and the encrypted bitstream. This is required to prove for the CSP that i) the scan report has not been compromised by the client after its generation, and ii) the scanned bitstream is the one encrypted afterward. Since the TEE is on the client side, the client provides as an input to the enclave the plain application bitstream and the secret key for encryption. After initiating the enclave, the CSP attests the enclave binaries to ensure their integrity. Then, the CSP exchanges a session key with the enclave to establish a secure link with the enclave. Through this secure link, the CSP sends a secret key for the computation of $PoI_{Enclave}$. At the end of step ②, the CSP receives the encrypted application bitstream and $PoI_{Enclave}$.

CSP side vs. client side: trade-offs. In terms of *performance*, it is more efficient to run the virus scan on the CSP side than on the client side, assuming the client has access to a TEE in the first place, due to the limited computing resources of the client. In terms of *security*, compromising the TEE on the client side could result in forging the virus scan report to label a bitstream with rogue primitives as a benign bitstream. This could pose serious threats on the CSP infrastructure or functional failure of the FPGA shell. Whereas for a compromised TEE on the CSP side, the bitstream confidentiality is no longer guaranteed. This could lead to problems for companies in which IP theft has a financial impact. In principle, the CSP and the client could negotiate where the scan should happen. In practice, however, the solution where the TEE resides on the CSP side is most likely to be adopted.

Trusted shell: CSP propriety infrastructure information. We assume the trusted shell to be open-sourced to assure clients the integrity of the deployed cryptographic primitives that will process their bitstreams on the cloud FPGA and the protected access to the configuration engine. Thus, the client does not need to attest the integrity of the trusted shell interfaces (PCIe core, DRAM controller, etc.), which might be considered as proprietary information for the CSP. To achieve the goal of this work while protecting the CSP proprietary design, the trusted shell can be split into an open-sourced part (the attestable part) by the FPGA vendor and a customized part whose configuration is done through the attestable part. The customized part, which contains the proprietary FPGA interfaces, is not revealed to cloud clients. However, the customized part can be scanned for viruses by a trusted party, similar to client bitstreams in step ②, and a hash of the customized part is published instead. This is to assure the clients that the shell does not contain rogue circuits spying on their logic. FPGA configuration data of the customized part, are replaced with their hash value during the computation of the reference attestation report in step ① and the proofs computed in steps ④ and ⑤.

VIII. Related Work

To the best of our knowledge, there are few solutions that tackle IP protection in cloud FPGAs, these are thoroughly discussed in [54]. We briefly discuss the most relevant work [29]–[31]. The schemes in [29], [31] rely on an initial bitstream configured on a cloud FPGA to enable the protection and the configuration of clients' IP bitstreams. The initial bitstream contains a cryptographic core for decryption of IP bitstreams using secret session keys, which are obtained through a key exchange protocol that deploys public key cryptography, e.g., RSA or Diffie-Hellman. In [29], RSA private key is embedded in the RSA core in the initial bitstream, thus, the confidentiality of the initial bitstream must be protected and this is achieved, e.g., by leveraging the hardened AES core in Xilinx FPGAs. For that, the authors propose the FPGA vendor to program the AES secret key before deploying the FPGA in the cloud. In [31], the authors assume that the FPGA vendor configures the FPGA with the initial bitstream prior to deployment in the cloud and that the FPGA is constantly powered, even during shipping to the CSP, to maintain its configuration. In [30] the authors leverage the hardened SRAM-PUF, elliptic core cryptography and AES cores on SmartFusion-2 FPGAs of Microsemi. However, partial reconfiguration of the FPGA fabric is not available in these FPGAs, therefore the client should design and implement the interfaces to the host, which is inconvenient for the CSP.

Unlike `TruFPGA`, these solutions attempt to solve one part of the problem that is IP protection for cloud clients under the assumption of untrusted CSP. while the CSP has no assurance that the encrypted bitstreams do not include rogue circuits. Moreover, they do not prevent unauthorized configuration of client's IP by the CSP, since the CSP can replay client's messages and freely load the encrypted bitstream after the client releases the FPGA.

IX. Conclusion

We presented, discussed, and demonstrated `TruFPGA` the first trusted configuration scheme for cloud FPGAs that i) protects the intellectual property of clients through the support of encrypted bitstreams and ii) enables the cloud service provider to check the client's application bitstream for rogue circuits that could damage or disable the FPGA or attack the integrity of the cloud infrastructure. With solving both security challenges in one feasible protocol, `TruFPGA` provides the means for practical FPGA TEEs, which not only allows more clients to use FPGA cloud services but also the processing of sensitive data on cloud FPGAs.

Acknowledgment. This work is co-funded by the DFG – SFB 1119 CROSSING/236615297, the Intel Collaborative Research Institute (ICRI-CARS), the BMBF and HMWK within ATHENE, the CyberSecurity Research Flanders with reference number VR20192203, and the UK National Cyber Security Centre through the project *rFAS* (grant agreement 4212204/RFA 15971). We also thank the Xilinx University Program for providing tools and boards.

References

[1] G. V. Research, "Field Programmable Gate Array market size, share & trends analysis report," April 2020. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/fpga-market

[2] Amazon AWS, "Amazon EC2 F1," https://aws.amazon.com/ec2/instance-types/f1/.

[3] Microsoft Research, "Project Catapult," https://www.microsoft.com/en-us/research/project/project-catapult/.

[4] A. Cloud, "FPGA-based Compute-Optimized Instance Families," https://www.alibabacloud.com/help/doc-detail/108504.htm, 2019.

[5] D. R. Gnad, F. Oboril, and M. B. Tahoori, "Voltage Drop-Based Fault Attacks on FPGAs Using Valid Bitstreams," in *IEEE FPL*, 2017.

[6] K. Matas, T. La, N. Grunchevski, K. Pham, and D. Koch, "Invited tutorial: Fpga hardware security for datacenters and beyond," in *ACM/SIGDA FPGA*, 2020.

[7] C. Beckhoff, D. Koch, and J. Torresen, "Short-circuits on FPGAs caused by partial runtime reconfiguration," in *IEEE FPL*, 2010.

[8] T. La, K. Pham, J. Powell, and D. Koch, "Denial-of-service on fpga-based cloud infrastructures - attack and defense," *IACR TCHES*, 2021.

[9] G. Provelengios, C. Ramesh, S. B. Patil, K. Eguro, R. Tessier, and D. Holcomb, "Characterization of Long Wire Data Leakage in Deep Submicron FPGAs," in *ACM/SIGDA FPGA*, 2019.

[10] I. Giechaskiel, K. Eguro, and K. B. Rasmussen, "Leakier Wires: Exploiting FPGA Long Wires for Covert-and Side-Channel Attacks," *ACM TRETS*, 2019.

[11] F. Schellenberg, D. R. Gnad, A. Moradi, and M. B. Tahoori, "An Inside Job: Remote Power Analysis Attacks on FPGAs," in *DATE*, 2018.

[12] M. Zhao and G. E. Suh, "FPGA-based Remote Power Side-Channel Attacks," in *IEEE S&P*, 2018.

[13] K. Matas, T. La, K. Pham, and D. Koch, "Power-hammering through glitch amplification - attacks and mitigation," in *IEEE FCCM*, 2020.

[14] S. Tian and J. Szefer, "Temporal Thermal Covert Channels in Cloud FPGAs," in *ACM/SIGDA FPGA*, 2019.

[15] F. Schellenberg, D. R. Gnad, A. Moradi, and M. B. Tahoori, "Remote Inter-Chip Power Analysis Side-Channel Attacks at Board-Level," in *IEEE/ACM ICCAD*, 2018.

[16] I. Giechaskiel, K. Rasmussen, and J. Szefer, "C3apsule: Cross-fpga covert-channel attacks through power supply unit leakage," in *IEEE S&P*, 2020.

[17] G. Dessouky, A.-R. Sadeghi, and S. Zeitouni, "SoK: Secure FPGA multi-tenancy in the cloud: Challenges and opportunities," in *IEEE EuroS&P, to be published*, 2021.

[18] J. Krautter, D. R. Gnad, and M. B. Tahoori, "Mitigating Electrical-level Attacks Towards Secure Multi-Tenant FPGAs in the Cloud," *ACM TRETS*, 2019.

[19] T. La, K. Mätas, N. Grunchevski, K. Pham, and D. Koch, "FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs," *ACM TRETS*, 2020.

[20] Intel, "Intel software guard extensions," https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[21] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[22] ARM, "ARM TrustZone technology," https://developer.arm.com/ip-products/security-ip/trustzone.

[23] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *USENIX Security*, 2016.

[24] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves," in *NDSS*. The Internet Society, 2019.

[25] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *ACM EUROSYS*, 2020.

[26] Bahmani et al., "CURE: A Security Architecture with CUstomizable and Resilient Enclaves," 2020.

[27] U. Rührmair, J. Sölter, F. Sehnke, G. Dror, J. Schmidhuber, and S. Devadas, "Modeling attacks on physical unclonable functions," in *ACM CCS*, 2010.

[28] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, "Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms," *IACR TCHES*, 2020.

[29] K. Eguro and R. Venkatesan, "FPGAs for Trusted Cloud Computing," in *IEEE FPL*, 2012.

[30] B. Hong, H.-Y. Kim, M. Kim, T. Suh, L. Xu, and W. Shi, "Fasten: An fpga-based secure system for big data processing," *IEEE Design & Test*, 2017.

[31] M. E. Elrabaa, M. Al-Asli, and M. Abu-Amara, "Secure computing enclaves using fpgas," *IEEE TDSC*, 2019.

[32] *Intel Stratix 10 Device Security User Guide*, Intel Corporation, 10 2020.

[33] IBM, "Data-in-use protection on IBM cloud using Intel SGX," https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx.

[34] Alibaba, "ECS bare metal instance," https://www.alibabacloud.com/product/ebm.

[35] Microsoft, "DCsv2-series VM now generally available from azure confidential computing," https://azure.microsoft.com/en-us/blog/dcsv2series-vm-now-generally-available-from-azure-confidential-computing/.

[36] M. Nasr, R. Shokri, and A. Houmansadr, "Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning," in *IEEE S&P*, 2019.

[37] J. Vliegen, M. M. Rabbani, M. Conti, and N. Mentens, "SACHa: Self-Attestation of Configurable Hardware," in *DATE*, 2019.

[38] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Fpga intrinsic pufs and their use for ip protection," in *CHES*. Springer, 2007.

[39] A. Wild and T. Güneysu, "Enabling sram-pufs on xilinx fpgas," in *IEEE FPL*, 2014.

[40] A. Maiti and P. Schaumont, "Improved Ring Oscillator PUF: An FPGA-Friendly Secure Primitive," *Journal of Cryptology*, 2011.

[41] Y. Hori, T. Yoshida, T. Katashita, and A. Satoh, "Quantitative and statistical performance evaluation of arbiter physical unclonable functions on fpgas," in *IEEE ReConFig*, 2010.

[42] M. Ender, A. Moradi, and C. Paar, "The unpatchable silicon: A full break of the bitstream encryption of xilinx 7-series fpgas," in *USENIX Security*, 2020.

[43] C. S. Intel, "Intel software guard extensions remote attestation end-to-end example," 2016.

[44] V. B. et al., "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018.

[45] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX fails in practice," https://sgaxeattack.com/, 2020.

[46] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: exposing the perils of security-oblivious energy management," in *USENIX Security*, 2017.

[47] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0LTpwn: Attacking x86 processor integrity from software," in *USENIX Security*, 2020.

[48] Intel, "Security center," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html.

[49] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Controlled physical random functions," in *IEEE ACSAC*, 2002.

[50] C. Herder, L. Ren, M. Van Dijk, M.-D. Yu, and S. Devadas, "Trapdoor computational fuzzy extractors and stateless cryptographically-secure physical unclonable functions," *IEEE TDSC*, 2016.

[51] *UltraScale Architecture Configuration*, Xilinx Inc., 07 2020.

[52] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in *USENIX ATC*, 2017.

[53] T. L. Kaspar Matas, "Fpgadefender," https://github.com/KasparMatas/FPGAVirusScanner, 2019.

[54] F. Turan and I. Verbauwhede, "Trust in FPGA-accelerated cloud computing," *ACM Computing Surveys*, 2020.