What is All the FaaS About? - Remote Exploitation of FPGA-as-a-Service Platforms

NITIN PUNDIR, University of Florida, USA
FAHIM RAHMAN, University of Florida, USA
FARIMAH FARAHMANDI, University of Florida, USA
MARK TEHRANIPOOR, University of Florida, USA

Field Programmable Gate Arrays (FPGAs) used as hardware accelerators in the cloud domain allow end-users to accelerate their custom applications while ensuring minimal dynamic power consumption. Cloud infrastructures aim to maximize profit by achieving optimized resource sharing among its cloud users. However, the FPGAs' reconfigurable nature poses unique security and privacy challenges in a shared cloud environment. In this paper, we aim to understand the interactions between FPGA and the host servers on the cloud to analyze FaaS platforms' security. We propose a vulnerability taxonomy based on the runtime attributes of the FaaS platforms. The taxonomy aims to assist the identification of critical sources of vulnerabilities in the platform in allowing focused security verification. We demonstrate the proof-of-concept by characterizing the potential source of vulnerabilities in the Stratix-10 FaaS platforms. We then focused on only one major source to perform more focused verification. The proof-of-concept is demonstrated by identifying the potential source of vulnerabilities in the Stratix-10 FaaS platforms. Then, to conduct more focused verification, we narrowed our focus to only one major source. It aided in the identification of several low-level software vulnerabilities. The discovered vulnerabilities could be remotely exploited to cause denial-of-service and information leakage attacks. The concerned entities have released software updates to address the vulnerabilities.

Additional Key Words and Phrases: FPGA-as-a-Service, cloud, device drivers, denial-of-service, information leakage

1 INTRODUCTION

High-performance computing (HPC) demand is growing exponentially and is expected to be valued at over 50 billion USD by the end of 2023 [5]. GPUs alone cannot provide a sustainable solution to this growing demand due to high power consumption and fixed architecture. FPGAs have always been considered a viable alternative to GPUs for testing and implementing computationally expensive algorithmic hardware designs such as artificial intelligence and deep learning networks [57]. With the recent rise in FPGAs demand [17], FPGAs from major vendors, such as Intel [4], and Xilinx [2], are increasingly being adopted and deployed on cloud infrastructures, commonly termed as FPGA-as-a-Service (FaaS). FaaS platforms allow cloud users to remotely access FPGAs and use reconfigurable hardware support to execute their software applications. The hardware can be customized to suit the software application needs in order to achieve maximum performance. The large-scale FPGA deployment has enabled efficient, and low-power FPGA-based cloud applications such as web search ranking [59], storage systems [43], and deep learning networks [57]. With the growing popularity of such applications, the demand for FPGA-based cloud-oriented hardware architecture and service solutions is anticipated to continue growing [17].

In FaaS infrastructure, the FPGA resources are shared among multiple users spatially or temporally to accommodate a large number of users with finite resources. The best examples of such business models are Amazon web services (AWS) F1 instances [3], Nimbix cloud services [11], and Accelize [1]. Prior work [30, 31, 34, 52, 54, 64] has shown that malicious FPGA designs can raise security and privacy concerns on the cloud. For example, a continuously running large cluster of ring oscillators (ROs) can bring down an entire FPGA by power starvation,

Authors' addresses: Nitin Pundir, University of Florida, Gainesville, USA, nitin.pundir@ufl.edu; Fahim Rahman, University of Florida, Gainesville, USA, fahim034@ece.ufl.edu; Farimah Farahmandi, University of Florida, Gainesville, USA, farimah@ece.ufl.edu; Mark Tehranipoor, University of Florida, Gainesville, USA, tehranipoor@ece.ufl.edu.

causing denial-of-service (DoS) attack [31]. Although previous work has justified the need for security protocols on cloud FPGAs, in most cases, the attacks were demonstrated on older FPGAs that do not resemble or are incompatible with commercial FaaS hardware platforms. The past work's primary focus was restricted to evaluating the security impact of malicious hardware designs. However, the security implications may also arise during interactions between the FaaS platform's various architectural components, such as the software stack defined by the host, device drivers, design development tools, embedded firmware, and the user design itself. The entire FaaS infrastructure is still emerging and undergoing rapid development. Therefore, a comprehensive analysis of the security of FaaS frameworks is essential.

With this work, we aim to highlight that FaaS platforms are vulnerable to more than just malicious hardware designs; an attacker may also take advantage of weaknesses in the FPGA's interactions with other cloud components. Therefore, we propose a taxonomy of runtime weaknesses to verify and identify security weaknesses in the FaaS platforms. We show the proof-of-concept by performing the security analysis on Intels' Stratix-10 FaaS platform. The analysis leverages the proposed taxonomy to identify software-assisted hardware vulnerabilities. The analysis helped to identify the presence of CWE-835 (infinite-loops), CWE-226 (sensitive information uncleared before reuse), CWE-476 (Null pointer dereferencing), and CWE-390 (Detection of error without action) vulnerabilities. These vulnerabilities can be exploited during runtime to cause DoS and information leakage attacks. The vulnerabilities (CVE-2019-11165 and CVE-2019-14604) identified in this work have been disclosed to the relevant Intel's product security team under coordinated vulnerability disclosure [32]. These vulnerabilities have been addressed in the security updates [8, 9].

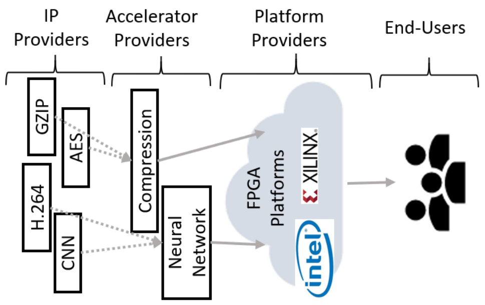

Fig. 1. FPGA-as-a-Service ecosystem highlighting different service providers.

The rest of the paper is organized as follows. Section 2 briefly explains background on FaaS architecture adopted by cloud vendors. Section 3 discusses the proposed taxonomy to identify runtime weaknesses in the FaaS platforms. Section 4 explains the runtime security analysis performed on Intel's FaaS platform and demonstrates the remote exploitation of security weaknesses. Section 5 discusses the challenges in analyzing the security implications of FaaS architecture. Section 6 discusses prior work on assessing FaaS security. Finally, Section 7 concludes the paper.

2 BACKGROUND

The cloud-based FPGA ecosystem has emerged to be known as "FPGA-as-a-Service" or "FaaS". Figure 1 shows the different entities involved in FaaS ecosystem. IP providers sell pre-compiled modules for various generic functionalities such as advanced encryption standard (AES), hash, Fourier filter, etc. [1, 2, 4]. The accelerator developer can then merge several of these IPs and develop a cloud-based accelerator module. The end-user can finally choose from these readily available hardware accelerators or develop its own to deploy them on the FaaS platform to accelerate their application. FaaS platform consists of FPGA connected to the host (generic server on a cloud) through a PCIe link. The end-user can remotely connect to the host and interact with the FPGA using vendor-provided APIs and utilities. In this section, we describe the FaaS architecture and its different components.

2.1 FPGA-as-a-Service Platform Architecture

The standard FaaS platform consists of three main components, i.e., the host, the physical FPGA, and the utilities. The utilities consist of tools, libraries, and drivers required to develop FPGA's hardware configuration files and facilitate efficient communication between the host and the FPGA. Figure 2 depicts a high-level overview of a generic (and the simplest) FaaS architecture that connects the host and the FPGA through virtualization and peripheral connection. The details of the architecture and the various steps involved are discussed as follows.

FPGA Setup and Initialization: At power-up, the initialization bitstream stored in flash memory configures PCI/PCIe hard-block, partial reconfiguration controller, and other required IPs, represented by steps 0 and 1 of Figure 2. This initial configuration of the FPGA is required for the host system to identify FPGA as a valid PCI/PCIe peripheral device and register it with a unique ID during PCI/PCIe enumeration [23]. If multiple FPGAs are connected to the same host, all are independently recognized and provided with distinctive device identifiers (IDs).

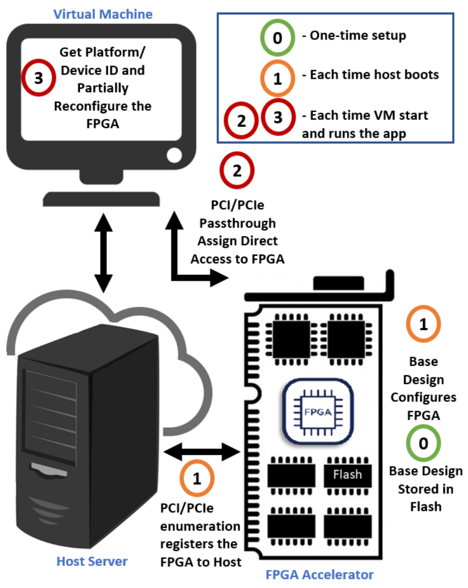
Virtual Machine
3
Get Platform/ Device ID and Partially Reconfigure the FPGA
0 - One-time setup
1 - Each time host boots
2 3 - Each time VM start and runs the app
2
PCI/PCIe Passthrough Assign Direct Access to FPGA
1
Base Design Configures FPGA
0
Base Design Stored in Flash
FPGA
Flash
1
PCI/PCIe enumeration registers the FPGA to Host
Host Server
FPGA Accelerator

Fig. 2. FaaS setup, initialization, and runtime flow.

Hardware Configuration File and Host Application: After the initialization phase, the peripheral FPGAs are visible and accessible through the host. The user configures the FPGA with the partial reconfiguration bitstream consisting of the custom hardware accelerator. On the host side, a user has his/her custom software application that interacts with FPGA for hardware acceleration, represented by step 3 of Figure 2. User's software application uses APIs to interact with FPGA. These APIs provide high-level abstractions of the communication protocols, which then invokes device drivers implementing low-level communication protocols to interact with the FPGA.

Security and Authenticity Checks: As mentioned earlier, the user's hardware accelerator is specifically a partial bitstream. This partial bitstream needs to be compliant with the FPGA's base design (Initialization bitstream used at power-up). Therefore, the FPGA's partial reconfiguration manager performs compatibility tests before programming the FPGA with the user's partial bitstream. The cloud vendor will also deploy additional security measures, such as shell logic and dynamic runtime monitors, to track the runtime behavior of the user's accelerator code [53].

Virtualization: The virtualization of PCI/PCIe devices on the cloud is a challenging task and causes significant drops in performance, thus defeating the purpose of hardware acceleration. One popular technique to overcome this bottleneck is PCI passthrough [61]. Hypervisors [25] on the cloud use PCI passthrough, which provides exclusive and direct access to the PCI/PCIe device from the host virtual machine to surmount performance drops.

2.2 Partial Reconfiguration and Multi-tenancy

FPGA is an integrated reconfigurable circuit consisting of configurable logic blocks, programmable routing interconnects, and other hard blocks such as digital signal processors (DSPs), memory blocks, transceivers, etc., to support any desired functionality and interface capability. FPGA development tools (such as Xilinx's Vivado [28] or Intel's Quartus [7]) translate the design implemented using hardware description languages (HDL) such as VHDL and Verilog into a gate-level netlist. The gate-level netlist, along with the constraint file containing the pin assignments for the FPGA, is provided to the development tool that translates, maps, and places-routes the final design following the FPGA design specification. The design configuration file (also known as a bitstream) generated from the last stage is provided to the FPGA reconfiguration controller (using JTAG, flash, PCI/PCIe) that configures the FPGA fabric according to the design specification.

However, if a user intends to make even a small change to the design, the entire process (which is very time-consuming) needs to be repeated. It can be inefficient when the hardware specification is large, and the user intends to replace only a specific module dynamically while preserving the rest of the FPGA's functionality. Partial reconfigurability solves this issue by splitting the design into multiple regions; fixed base region and PR regions. The PR regions could be dynamically replaced while keeping the base design fixed. PR technique plays a vital role in FaaS platforms. It allows the user to generate its hardware accelerator as PR design and program the FPGA dynamically while keeping the rest of the FPGAs functionality needed to communicate with the host remains intact.

At the time of writing the paper, due to security reasons, cloud vendors only allow temporal sharing of the FPGA resources, meaning only one user has access to the FPGA at a time. However, with advances in PR techniques to provide more isolation between PR regions, spatial sharing of FPGA resources on the cloud is possible soon. It would allow multiple users to have their hardware accelerator code running in their specific regions of the FPGA.

Table 1. Vulnerability taxonomy based on runtime attributes of FaaS platform.

Causes	Attributes	Effects
• Improper initialization of base design. • Improper initialization of environment variables (CWE-665). • Missing initialization of variables and resources (CWE-456, 909).	Setup/ Initialization	• Invalid initial memory configurations. • Incorrect initial state. • Undiscovered peripheral device. • Unexpected branch resolution due to bad initial state.
• Vulnerable HLS translation. • Bad coding practices. • Violating secure coding guidelines (CWE-1006).	Development	• Information leakage due to improper pipeline balancing. • Flow hijack due to insertion of vulnerable control paths. • Insecure memory translations.
• Out-of-bound read/writes. • Untrusted pointer dereference. • Uninitialized pointer access. • Incoherent micro-architectural state between host and FPGA.	Runtime and Memory Accesses	• Unauthorized privileged memory access. • Memory leakage/corruption. • Erroneous computations due to stale translations or invalid block accesses. • Crashing/slowdown of the entire system.
• Bad implementation of interrupt service routine (CWE-430, 431, 432, 479, 248, 703).	Exception/ Interrupts	• Denial-of-Service due to abrupt system failures. • Race-conditions and deadlocks. • Information loss/leakage due to ignored interrupts.
• Dynamic power consumption. • Dynamic timing information.	Side-channel	• Side channel information leakage.
• Incomplete cleanup on exit (CWE-459). • Not failing securely on system failure (CWE-460).	Termination	• Leakage of residual logic. • Resources unavailable to other accelerators.

3 VULNERABILITY TAXONOMY

This section provides a comprehensive taxonomy for threat assessment of FaaS platforms based on the runtime attributes. FaaS platforms are a symbiotic association between the reconfigurable hardware and the remote host machine. The host machine is a primary remote contact with the user, and the access to the FPGA is abstracted with the help of APIs and tools. Due to the critical involvement of software interactions, the provided taxonomy highlight both software and hardware-based weaknesses. Some of these weaknesses are mapped to the common weaknesses (CWE[1]). Security challenges that can arise at each stage of the FaaS runtime attribute are briefly summarized in Table 1. Some of these weaknesses can only be seen in spatial sharing scenarios that cloud vendors do not currently provide. We assume spatial sharing will be soon possible due to advances in PR technique providing secure logical isolation and FPGA-based operating systems [62]. Thus, spatial sharing should also be considered when constructing a security taxonomy for FaaS platforms.

Setup/Initialization: The process of setup/initialization refers to the period between power-up and the state when the device is available to the end-user to accelerate its application. It also constitutes storing of initialization bitstream in FPGA's flash, programming FPGA with the user's PR files, setting up the host with the necessary APIs/tools/drivers, and transfer the control from one virtual machine to another using PCI passthrough.

In this phase, security vulnerabilities can arise due to improper initialization of the base design, tools, or other environment variables (CWE-665), missing initialization of a variable or resources (CWE-456,909), etc. These weaknesses can trigger invalid memory configurations that can lead to violation of memory permissions [63]. In worst-case scenarios, the peripheral device can remain undiscovered to the host or lead to errors due to invalid PR controller configurations. It can also cause program drivers to take unexpected branches due to reliance on the initial null/'0' value of a resource.

Development: The user's accelerator design files are generated offline due to the long compilation time taken by FPGA synthesis tools. The tools could also use the HLS compiler to translate the accelerator's HLL code specification to an intermediate HDL representation. Most of the interface modules used during the accelerator's compilation are pre-built and are used as a plug-n-play style to reduce design complexity. These development tools themselves can unintentionally introduce vulnerabilities arising from code reuse, dead codes, etc., [27]. Due to large development teams, increased complexity of the systems, following bad coding practices (CWE-1006), and violating secure coding guidelines (CWE-1006), the bugs could be buried deep into the tool's source code.

Similarly, HLS can introduce vulnerabilities [49–51] during translation from an untimed and sequential C code to a concurrent HDL. For example, improper balancing of pipelines or insertion of vulnerable control paths by HLS can lead to information leakage or make the design susceptible to fault injection attacks. The HLS translations can also cause insecure memory mappings. For instance, to conserve expensive resources like registers and local memory, the compiler can map critical data vectors to the FPGA's global memory (external DRAM). The global memory is tagged as insecure because it can be shared among multiple FPGA accelerators and is also accessible by the host application [48].

Runtime and Memory Accesses: There are different types of memory accesses across the FaaS platform. The host can read and write to FPGAs' global memory, while the user's accelerator can read and write to the FPGA device's internal and external memory. Such systems should be evaluated for security threats caused by out-of-bound reads/writes (CWE-125,787), untrusted pointer deference (CWE-822), and uninitialized pointer access (CWE-824) to avoid confidentiality and integrity violations. Likewise, memory leakage and unauthorized memory accesses across FPGA accelerators can lead to incorrect results and privacy breaching, or lead to DoS attacks by one accelerator spending all the global resources of the FPGA [45].

[1]CWE is a community-developed list of common software security weaknesses. Visit https://cwe.mitre.org for more information

As per the PCIe specification [12], anyone on the bus can read and write transaction layer packets (TLPs). Therefore, restrictions must be in place to prevent any malicious FPGA accelerator from accessing the host privileged memory space or sniffing around other devices' TLP packets. It is crucial in the cloud scenario, where multiple FPGAs and devices used by various users are connected to the same host. Since FPGAs can perform low-level hardware transactions between the FPGA and the rest of the system, a malicious FPGA accelerator can execute illegal transactions to the system bus if restrictions are not in place. This uninterrupted access can result in information leakage, unauthorized privilege escalation, memory corruption, and a system failure [22, 45, 46].

Similarly, due to the micro-architectural system state between the host and FPGA, the architecture should be evaluated for security vulnerabilities such as memory coherence requests [24]. For example, the use of *get_user_pages()* for DMA in a heterogeneous accelerator using non-RAM based file-system can cause host's kernel crashes [26]. Coherence vulnerabilities can also result in erroneous computations due to stale translations or invalid block accesses. They can even result in the whole system's slowdown causing a performance hit [60].

Exception/Interrupts: Interrupts are a crucial part of a system's core functionality and effectively inform processors about a high-priority task. An interrupt service routine (ISR) is invoked to handle the respective interrupts. The deployment of a wrong handler (CWE-430), missing a handler (CWE-431), a dangerous signal handler that is not disabled during sensitive operations (CWE-432), and use of a non-reentrant function signal handler (CWE-479) can result in vulnerabilities causing information leakage, DoS attacks, race-conditions, or deadlocks. For example, the accelerator is in an unknown state after power-up until the interrupt-handler is set up in the memory-mapped device (MMD) layer. Any prior interrupt from the kernel would be lost, causing information or data loss if the interrupt was to signal the accelerator to start reading data from the host application.

Accordingly, the system should efficiently handle runtime FPGA exceptions. Due to the hosting of other services, cloud deployment conditions may differ from FPGA libraries and drivers' development and testing conditions. The exceptions and errors can arise due to incoherency, and the FaaS system should handle them and "fail-safe" or exit securely. Uncaught exceptions (CWE-248) or improper check or handling of exceptional conditions (CWE-703) can cause abrupt system failures or sensitive information exposure. For example, a failure to get DMA lock of user-pages exception should be handled safely with proper termination. Otherwise, this could be exploited to carry out DoS attacks as shown in Section 4.2.

Side Channel: The FPGA accelerators' runtime properties constitute the side-channel information and can contribute to the side-channel leakage. For example, previous work has demonstrated that dynamic power consumption of the FPGA can be used to reveal the secret key of the cryptographic engine running on the same FPGA [37, 38, 52]. Similarly, the side-channel timing information of the FPGA accelerator, host application, or any other module on the FaaS could leak sensitive information. Kocher et al. [38] was the first to open this class of attacks by demonstrating timing attacks to leak Rivest Shamir Adleman (RSA) keys. Recently, vulnerabilities in out-of-order execution (OOE) architectures gained much attention in the security community. Speculative execution attacks, such as spectre [36], meltdown [41], foreshadow [58], MDS [56], etc., have been successfully demonstrated on host systems. The FPGA accelerators may also benefit from similar speculative executions to improve the performance, raising similar security implications.

Termination: Termination can be described as an FPGA state when the accelerator returns to the idle state after successful execution or exits prematurely due to a failure. It is crucial to ensure that the accelerator is systematically and securely terminated in both cases. The protocols should be in place to clear the FPGA global memory and any residual logic of the previous accelerator. Incomplete cleanup (CWE-459) or improper cleanup on thrown exception (CWE-460) can result in information leakage or resources being unavailable to other accelerators [40, 44]. As shown in the Global Memory Leakage vulnerability in Section 4, the residual data of the FPGA accelerator remained in the FPGA's global memory, can be exposed by the malicious FPGA accelerator.

4 SECURITY ASSESSMENT OF INTEL'S STRATIX-10 FAAS PLATFORM

This section discusses using the proposed taxonomy to identify the vulnerabilities in Intel's Stratix-10 FaaS platforms. Based on the attributes discussed in the taxonomy, we classified the runtime behavior of running a program on the Stratix-10 FaaS platform as follows:

- Setup/Initialization: The FPGA vendor provides the base design, which flashes the FPGA and initializes the PCIe hard block and other components needed to communicate with the FPGA from the host.
- Development: On the host side, Intel FPGA OpenCL software development kit (OCLSDK) [21] facilitate the development process. OCLSDK consists of utilities and libraries essential for developing FPGA accelerators specified in OpenCL and HDLs using the utility 'aoc', also known as 'offline compiler'. The utility uses HLS compiler [19] to translate OpenCL FPGA accelerator specification to an intermediate RTL specification and Quartus Prime tool to compile PR configuration files of the FPGA accelerator.
- Runtime and Memory Accesses: Intel FPGA runtime environment for OpenCL (OCLRTE) [18] facilitate the runtime communication between the host and the FPGA. OCLRTE enables the execution of pre-compiled FPGA accelerators on the targeted FPGA board. It typically consists of utilities for high-level tasks, host-runtime libraries, drivers, and runtime environment-specific libraries. OCLRTE utility, "aocl", supports high-level tasks such as configuring the host-side development flow, managing FPGA boards, managing host applications, etc.
- Exception and Interrupts: The exceptions and interrupts could be raised by the software or the hardware peripherals (FPGA). Software interrupts are largely handled by the APIs, whereas device drivers facilitate handling hardware interrupts from the FPGA.
- Side-Channel: Side-channel exploitation largely depends on the capabilities of an attacker to remotely analyze the timing and power side-channel information of the hardware accelerator running on the cloud. As per our knowledge, these capabilities were very minimal and inaccurate to carry out any successful remote side-channel attack.
- Termination: After the acceleration has been successfully terminated, the user may either program the FPGA with a new accelerator binary or relinquish control of the FPGA to allow it to be allocated to another user. The user's accelerator binary can only configure the FPGA's reconfigurable region; the newly programmed accelerator binary has no impact on the FPGA's peripheral components such as external memory.

We were able to classify critical sources of vulnerabilities in the Stratix-10 FaaS platform using the above characterization based on the proposed taxonomy. These sources are:

(1) 'aoc' and 'aocl' utilities: High-level utilities allow a user to interact directly with the FPGA. Since they can invoke privileged system calls, vulnerabilities in these utilities can be crucial.
(2) Device Driver: This is low-level software running at the highest privilege and facilitates communication between the host and FPGA.
(3) Quartus and HLS: Vulnerabilities in these development products could introduce vulnerabilities in the generated hardware accelerator binary.

To concentrate our verification efforts, we focused only on the security verification of device drivers. After analyzing the device driver code for Stratix-10, we divided it into four major functionalities it performs to support the host application.

(1) Programs the FPGA with user's custom accelerator binary.

(2) Facilitate the transfer of data from the host's memory to FPGA's external memory.
(3) Facilitate transfer of computed data from FPGA to the host.
(4) Frees up any allocated memory using the transfers.

From the device driver [2], we isolated the source code for each of the above functionality. We took advantage of various static analysis tools to identify possible weaknesses in the isolated source code. Most of the identified weaknesses were redundant and could not be exploited. However, some of them could be exploited from the userspace. The identification of vulnerabilities and the generation of exploitable test cases was a lengthy and exhaustive process. In the rest of the section, we discuss the details and in-depth analysis of the vulnerabilities exploitable from the userspace.

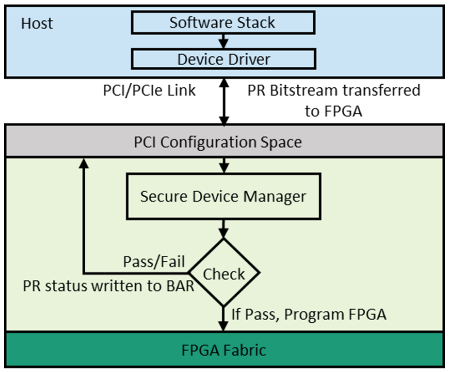

Fig. 3. Sequence of steps occurring after triggering of PR by the host application or "aocl" utility.

Experimental Setup: The experimental setup replicates the commercial FaaS platforms, such as Nimbix[11] which uses the Terasic DE-10 Pro Board [14]. Direct access to the commercial FaaS platform was restricted due to the nature of our work. The initial setup of the FPGA device and the tools are done in accordance with the vendor specifications and guidelines [21]. The FPGA is initially loaded with the vendor-provided initialization bitstream so that it is recognized as a PCI/PCIe compatible device by the host during boot time. The host runs Ubuntu 16.04 LTS, Intel FPGA OpenCL SDK [21], and Quartus Prime Pro 18.1.1/19.01 [18].

4.1 Partial Reconfiguration DoS Vulnerability

FPGA programming with the user's accelerator is initialized by transferring the pre-compiled accelerator binary from the host to the FPGA's partial reconfiguration manager (Secure Device Manager (SDM) in the case of Stratix-10) via the PCI/PCIe link. The FPGA requires the device drivers to facilitate this transfer. The device driver runs at kernel privilege-level, although the host application runs at the user level. It makes the device driver's vulnerabilities critical if the user can exploit those from the userspace.

The analysis of the isolated source code of functions responsible for sending the partial bitstream to FPGA identified the usage of uninterruptible infinite loops as shown in Listing 1 Line 4. The loops were used to wait indefinitely for the SDM to successfully program the FPGA because the bitstreams' size may differ for different accelerators. However, it is not a reasonable supposition that the SDM will always successfully program the FPGA. Therefore, these infinite loops can be exploited to trigger a DoS attack.

Listing 1. Vulnerable code in PR module.

```
1 // Wait for PR complete
2 status = ioread32(aclpci->bar[ACL_PRCONTROLLER_BAR]+ACL_PRCONTROLLER_OFFSET+ALT_PR_CSR_OFST)
      ;
3 ACL_DEBUG (KERN_DEBUG "ALT_PR_CSR_OFST status is 0x%08X", (int) status);
4 while(status != ALT_PR_CSR_STATUS_PR_SUCCESS)
5 {
6   msleep(100);
7   status = ioread32(aclpci->bar[ACL_PRCONTROLLER_BAR]+ACL_PRCONTROLLER_OFFSET+ALT_PR_CSR_OFST
      );
```

[2]The source code for the Stratix-10 device driver is publically available at Intel's website [6]

```
8  ACL_DEBUG (KERN_DEBUG "ALT_PR_CSR_OFST status is 0x%08X", (int) status);
9 };
```

We showed that these infinite loops could be used to deploy DoS attacks during the partial reconfiguration process. In Stratix-10, the FPGA fabric is partially configured using Configuration-via-Protocol (CvP) [16] over the PCI/PCIe, which can be initiated using ”aocl” utility or by using OpenCL APIs. As shown in Figure 3, the system performs the following series of actions upon PR initialization.

(1) Host-side ensures the compatibility of PR files with the base design.
(2) To prepare for partial reconfiguration, PR control signals are asserted, which disables the PR region and all its outputs.
(3) PR configuration file is sent to SDM over PCI/PCIe, which configures the PR after compatibility checking and writes the status back to PCI/PCIe base address register (BAR).
(4) The system waits for the 'SUCCESS' signal, and the normal operation is continued after that.

The cloud gives its users no control over the compatibility check as it remains enabled in the compiler and SDM firmware [20]. However, any vulnerability in the compatibility checker creates functional and security issues. As seen in line 4 of the snippet, the while loop waits for the 'SUCCESS' status state on the PCI/PCIe BAR. If an attacker can deliberately activate a false state, the driver will remain inside the loop. Since drivers run in a kernel context, meaning that the resource cannot be reclaimed from the user domain, such an action will cause the driver to be stuck indefinitely, rendering FPGA inaccessible to other users.

To successfully trigger a DoS attack, the adversary must bypass the host-side compatibility check and manage to deliver the incompatible bitstream to the SDM. For our demonstrated attack, we used the vendor-supplied BSP compiled using version 18.1.1 Build 263 and generated our PR bitstream using version 18.1.1 Build 277. The two builds had the same functionality. We were able to successfully deliver the bitstream to SDM using the "aocl" utility. Since the PR bitstream was slightly inconsistent with the base design, SDM failed to program the FPGA and wrote back 'FAIL' status to the BAR, triggering the vulnerability and causing the driver to get stuck in the non-interruptible loop. An attacker can exploit the vulnerability on the cloud to cause a DoS attack by programming FPGA with PR bitstream incompatible with the base design (such as incorrect voltage levels or signal values) but close enough to pass the soft checks. FPGA resources remain unavailable until the host is rebooted (frequent rebooting could be challenging due to shared server resources between multiple users). It may result in financial losses to the cloud due to the unusability of (unavailable) resources.

4.2 DMA Page Lock Failure

This vulnerability is related to the data transfer process between the host application and the Stratix-10 FPGA. In this case, the host initiates direct memory access (DMA) to bypass the CPU to achieve high transfer bandwidth. The FaaS platform's detailed overview of the DMA is discussed below.

Address Space and DMA: Virtual address space offers any running process a 'false' sense of execution that it can access the entire memory space. The virtual addresses are translated to the main memory's physical addresses by the joint effort of the memory management unit (MMU) and the CPU. Usually, access to the main memory is supervised by the CPU, which results in significant performance overhead. DMA feature allows certain hardware subsystems to directly access main memory bypassing the CPU and offloading excessive memory operations to a dedicated DMA engine. The DMA module of the driver performs the following series of steps to support DMA between the host and FPGA:

(1) It translates the userspace virtual addresses (of user-data to be transferred) to the physical addresses and pins them to the memory to prevent modification, movement, or removal from the memory during the transfer.

(2) The memory accesses (e.g., memory to memory copying) are offloaded to the dedicated DMA engine.
(3) The pinned pages are marked as dirty upon DMA's completion, and their references are removed.

A control flow graph (CFG) was generated for the isolated DMA functionality code to study the function calls. The CFG analysis showed that the aclpci_dma_update() function in the device driver was crucial to handling all of the DMA requests. From this function the subsequent call to the lock_dma_buffer() function is responsible for getting user pages and pinning them to the main memory. To do this the called lock_dma_buffer() function invoked the system call get_user_pages_remote(). We noticed that the module missed an error handler if the system call is unsuccessful in pinning the user pages. It can cause a DoS attack from the userspace because drivers will have no routine to handle that error, and no one can force drivers to relinquish the resources. To exploit the vulnerability, we modified (to support the current platform) the implementation of OpenCL AES [35] to transfer large chunks of data in memory, expecting the driver to crash in the pinning stage of the memory.

Listing 2. Kernel output for DMA page lock vulnerability.

```
 1 [  330.375868] aclpci_close (225):
 2 [  330.375870] aclpci = 00000000d0cd263a, pid = 2680, dma_idle = 1
 3 [  346.182065] aclpci_open (167):
 4 [  346.182067] aclpci = 00000000d0cd263a, pid = 2716 (Benchmark-OpenC)
 5 [  346.182111] init_irq (407):
 6 [  346.182111] using a 64-bit irq mask
 7 [  347.132044] lock_dma_buffer (331):
 8 [  347.132045] Couldn't pin all user pages. -14!
 9 [  347.132046] aclpci_dma_update (715):
10 [  347.132046] Failed lock dma buffer for 16384 bytes
```

As one can see in line 8 of the Listing 2, during our example exploitation run, the kernel could not pin all the user pages during our example exploitation run, and the driver returned the error code -14. The vulnerability arises because the caller (the parent function calling lock_dma_buffer()) inherently assumes that the system call will always be able to pin pages to the memory successfully. Consequently, when the call returns with an error, there is no handler to take action (CWE-390), causing the kernel to crash, inevitably leading to a DoS attack. As one can see, a similar attack can be easily performed over cloud architecture by a host application attempting to transfer large chunks of data resulting in a remote DoS attack.

4.3 Global Memory Leakage

This vulnerability demonstrates FPGA's global memory leakage, which is shared between the host and the FPGA. Different types of memory resources are available on the FaaS platform, as shown by the memory hierarchy in Figure 4. The main memory resides on the host and is local to the host process. On the other hand, the "Local" and "Private" memory consisting of block RAMs and registers reside on the FPGA fabric and are local to the FPGA device. The global memory is an external memory situated on the FPGA platform, shared between the host and the FPGA to facilitate data transfer between them using DMA.

In Stratix-10, peripheral resources such as external memory, GPIOs, I/O modules, etc., must remain as a part of the base design [10]. Therefore, the FPGA's global memory is not configured by the partial reconfiguration bitstream (user's accelerator). It could cause the residual data to remain in the memory, which could then be leaked by the new user to whom the FPGA is assigned in the cloud.

To exploit this vulnerability, we created a simple malicious FPGA accelerator as shown in Figure 4 box 6 to read out the data used by the previous FPGA accelerator. The malicious accelerator allocates two buffers, A and B, in the FPGA's global memory. It copies buffer A's contents to buffer B, which is read back by the host application. Since space for buffer A is allocated but not initialized to any value by the host application, it simply copies the previous accelerator's residual data.

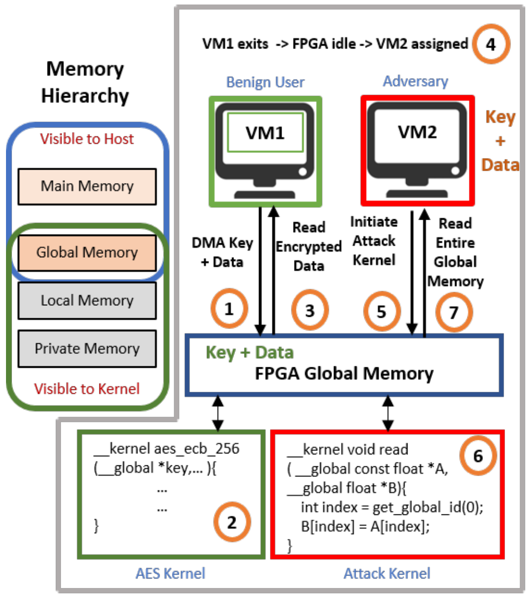

Fig. 4. Memory hierarchy for FaaS platform and the procedure of global memory leakage attack.

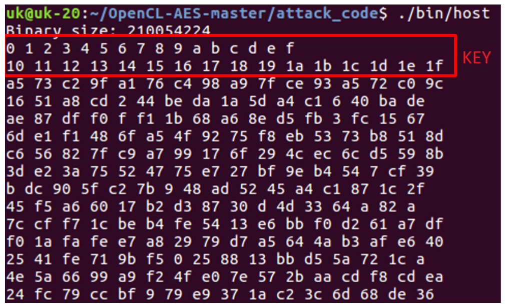
```
uk@uk-20:~/OpenCL-AES-master/attack_code$ ./bin/host
Binary size: 210054224
0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
a5 73 c2 9f a1 76 c4 98 a9 7f ce 93 a5 72 c0 9c
16 51 a8 cd 2 44 be da 1a 5d a4 c1 6 40 ba de
ae 87 df f0 f f1 1b 68 a6 8e d5 fb 3 fc 15 67
6d e1 f1 48 6f a5 4f 92 75 f8 eb 53 73 b8 51 8d
c6 56 82 7f c9 a7 99 17 6f 29 4c ec 6c d5 59 8b
3d e2 3a 75 52 47 75 e7 27 bf 9e b4 54 7 cf 39
b dc 90 5f c2 7b 9 48 ad 52 45 a4 c1 87 1c 2f
45 f5 a6 60 17 b2 d3 87 30 d 4d 33 64 a 82 a
7c cf f7 1c be b4 fe 54 13 e6 bb f0 d2 61 a7 df
f0 1a fa fe e7 a8 29 79 d7 a5 64 4a b3 af e6 40
25 41 fe 71 9b f5 0 25 88 13 bb d5 5a 72 1c a
4e 5a 66 99 a9 f2 4f e0 7e 57 2b aa cd f8 cd ea
24 fc 79 cc bf 9 79 e9 37 1a c2 3c 6d 68 de 36
```

Fig. 5. Residual from previous AES accelerator containing secret key.

To show that vulnerability can be exploited to leak sensitive information, we utilized the OpenCL AES implementation of the OpenSSL engine [35]. We did the encryption and relinquished control of the FPGA. We then reassigned FPGA and reprogrammed it with our malicious accelerator to dump the global memory. By adjusting the sizes of buffers A and B in the host program, we successfully dumped the entire global memory, which contained plaintext, ciphertext, and the encryption keys, as shown in Figure 5. Following are the series of steps taken to execute the attack, also depicted in Figure 4.
❶ AES host-side software application initiates AES FPGA accelerator and transfers (DMA) encryption key and plaintext to the global memory of FPGA.
❷ AES FPGA accelerator performs the encryption on plaintext using the provided key and writes the encrypted text (ciphertext) back to the global memory.
❸ AES host application reads the ciphertext from the FPGA's global memory.
❹ The AES application user no longer needs the FPGA and releases it for other users on the cloud. The cloud assigns the physical FPGA resource to another user (which may be the adversary in our example attack case).
❺ The adversary launches its host application as well as the malicious FPGA accelerator.
❻ The malicious accelerator reads the residual data of the previous process from the global memory and copies them to a separate buffer on the global memory.
❼ The adversary's host application reads back the collected values and analyzes them to reveal any secret keys or sensitive information.

We tested whether the same scenario persists on the Xilinx FaaS platforms. We found that the Xilinx's compilation tool creates two accelerator bitstreams [acclerator_name].bit and [accelerator_name]_clear.bit. The clear bitstream is loaded before programming the FPGA with any other accelerator to clear all the memories and residual logic [15]. Therefore, we could not replicate the scenario on the FaaS platform using Xilinx's accelerator board.

4.4 NULL Pointer Dereferencing

A NULL pointer dereference exception occurs when an application dereferences an object that is expected to be valid but is NULL [33]. It typically causes the segmentation fault or memory violation and is managed effectively in advanced languages such as Java and C++. However, in C, which is used for low-level system implementations, NULL is a built-in constant that evaluates to '0'. *The challenge is that an x86 systems also contain a valid address '0' in the kernel address space.* Hence, if a user could trick the kernel into reading and writing from/at address '0', the user can effectively increase its privilege-level to run tailored exploits from a higher privilege that is not available in the regular (constrained) user mode.

Listing 3. Null Pointer De-referencing kernel buffer error trace.

```
[  149.493770] BUG: unable to handle kernel NULL pointer dereference at 0000000000000070
[  149.493833] Call Trace:
[  149.493836]  aclpci_release_user_pages+0x2b/0xc0 [aclpci_de10_pro_drv]
[  149.493838]  unlock_dma_buffer+0x2e/0x90 [aclpci_de10_pro_drv]
[  149.493840]  aclpci_dma_update+0x27c/0x650 [aclpci_de10_pro_drv]
[  149.493841]  wq_func_dma_update+0x17/0x20 [aclpci_de10_pro_drv]
[  149.493844]  process_one_work+0x14d/0x410
[  149.493846]  worker_thread+0x4b/0x460
[  149.493848]  kthread+0x105/0x140
[  149.493849]  ? process_one_work+0x410/0x410
[  149.493851]  ? kthread_destroy_worker+0x50/0x50
[  149.493852]  ret_from_fork+0x35/0x40
```

Implementing and accessing a completely separate space is quite expensive, due to which all operating system implementations embed the kernel space in the user's process address space and rely on page protections to prevent the user from accessing it [42]. As a result, when the system switches to the kernel mode, the kernel pages become accessible. The pages associated with the user's process are also visible and accessible to the kernel. The system can be switched to kernel mode by invoking system calls which serve as a set of entry-points for the userspace to interact with the kernel. In a nutshell, it will try to access page zero for the case where the kernel dereferences the NULL pointer. Therefore, if the user (adversary) can map a page to page zero (usually using system-call mmap) and induce the kernel to dereference a NULL, it will make the kernel getting user space data where the attacker controls the NULL dereferenced information and performs the code injection attack.

From our previous case studies, we learned that the large memory allocation for DMA or the pinning of large user pages to memory could trigger vulnerabilities. As a consequence, certain user pages can ultimately have Null pointers. On further review of the device driver revealed that the function responsible for releasing user pages does not test whether or not the pages are Null. Therefore, it can be used to trick the kernel into dereferencing NULL pointers.

We exploited this vulnerability by creating large buffers in the host application. These buffers were not initialized in an attempt to cause certain user pages to Null. If the user pages were not pinned to the memory, the pointers to those pages were assigned as NULL. As a consequence, when DMA module calls aclpci_release() to release pinned pages (as explained previously in DMA page lock vulnerability), it ends up dereferencing NULL pointers. The detailed error trace is provided in Listing 3.

NULL pointer dereferencing vulnerability leads to kernel crashes causing DoS. When combined with a memory mapping attack (discussed earlier), it can be used for remote code injection attacks. However, beginning with Linux Kernel 2.6.23, mmap_min_addr is implemented to prevent against such types of memory mapping attacks. It is a kernel tunable (can be set by the system admin) that prevents new memory mappings below the specified minimum address by unprivileged users (with the default being 0-4096 address range). However, literature shows that such countermeasures can be bypassed [55], exposing the system to the risk of NULL dereferencing exploits. These vulnerabilities have been patched, and as per our knowledge, there are no known publicly disclosed attacks that circumvent mmap_min_addr() securities.

5 DISCUSSION

In previous sections, we focused on the emerging architecture of FaaS, presented a taxonomy of vulnerabilities in FaaS platforms, and performed the security analysis on Intel's FaaS platform. The distinctive characteristics of FaaS architecture reveal a diverse set of security implications and offer a plethora of opportunities to develop comprehensive security guidelines and runtime device and system-level countermeasures. However, building a secure FaaS architecture against numerous existing and potential future hardware and software security exploits also exhibits a unique set of challenges and calls for well-guided collaborative research from FPGA vendors, cloud service providers, and security researchers.

The researchers' significant challenges in this domain arise from some of the fundamental restrictions and anonymity of the commercially deployed architecture. Although it is difficult to point out and solve each challenge, we note some of the major obstacles that hinder the research and development of security and privacy-aware FaaS platform architecture.

- The primary challenge for developing security-aware FaaS architecture is that the vendors and the developers take performance-oriented features such as speed, power, and throughput, as their focal point. Design and development parameters relevant to such an architecture's security are at par with the traditional usage and not fully integrated into the FaaS architectures and services. Security versus performance trade-off is also not adequately established based on some standard security metric.
- Security attributes for FaaS architecture are not well-established among the cross-community researchers. Additionally, to the best of our knowledge, there lacks a standard security metric to access the FaaS infrastructures and currently solely relies on datacenters' internal security assessments. The FPGAs are often integrated into the already existing infrastructure, and also the FaaS infrastructure can vary from one provider to another. We need to establish standard security policies and properties common for different infrastructures hosting reconfigurable hardwares.
- Apart from the data centers' customized implementation, the FaaS architecture is also highly dependent on the device vendor. The device vendor is solely responsible for providing the base communication APIs and tools to communicate and generate applications for their device. The security implications of such APIs and tools should be well established and analyzed.
- Most security assessments and proposed countermeasures in the literature still rely on the traditional academic settings in the laboratory environment. The primary reason behind this is the anonymity maintained by the data centers and the lack of collaboration between industry and academic security researchers. The anonymity leaves room for researchers to make false assumptions while assessing the security implications.

6 RELATED WORK

Although a growing number of academic researchers advocate the need to address the security challenges in the FaaS architecture, existing work does not always consider the FaaS hardware platform as a symbiotic relationship between software and hardware. Schellenberg et al. [52], and Krautter et al. [39] demonstrated that on-chip power monitors in an FPGA could be used to leak secret from the encryption engine running in distinct sectors of the same FPGA. Iakymchuk et al. [34] leaked AES secrets using key data heat patterns and successfully demonstrated that temperature could also be used as a covert channel for secret leakage. Muhtaroglu et al. [47] observed that if a voltage drop in the FPGA's power distribution network was high enough and sustained for a considerably longer time, it could lead to a voltage emergency. Gnad et al. [31] used this technique to orchestrate a DoS attack on remote FPGAs. Tian et al. [54] showed that cloud-based FPGAs take some time to cool down and can be used as a covert channel to disclose prior user's FPGA state, demonstrated on cloud FPGAs of Texas Advanced Computing Center [13] for academic research. This work also relies on ROs to raise the FPGAs temperature. Giechaskiel et al.[30] utilized long FPGA routing wires bearing logical '1' as covert leakage channel. The attack relied heavily on the design circuit's placement information and manual placement of the IP cores, which is not easy to obtain in the commercial FaaS architecture scenario.

In a different context, Frisk et al. [29] showed that inexpensive FPGA hardware devices linked to the PCI/PCIe slot of the host could be used to dump memory and, therefore, to increase privilege levels. However, this technique relied on the physical access of the system. Furthermore, similar vulnerabilities exist in the SoC-FPGA domain where processor and FPGA are built on the same die, as outlined by Chaudhuri et al. [24].

These research findings provide a sound foundation for assessing FaaS security. The majority of works utilize implementing logical ring oscillators (ROs) or some combinational loops to increase the on-chip temperature or cause voltage starvation (drop) to assist their attacks. However, commercial cloud vendors often scrutinize such circuitry in the user design and restrict using them as blacklisted circuits [53]. Moreover, some attacks were demonstrated on primitive FPGA boards. The high-end FPGA accelerators deployed on current FPGA platforms are generally more resilient to voltage, current, temperature fluctuations.

7 CONCLUSION

We reviewed the architectural and functional aspects of the practical FaaS platform. After analyzing interactions between various components of the FaaS platform, we presented the vulnerability taxonomy associated with the platform's runtime attributes. We then demonstrated the feasibility of remote attacks on FaaS platforms by doing the security analysis of Intel's Stratix-10 FaaS platform. The analysis revealed security vulnerabilities in the device drivers, which an attacker could exploit to cause remote DoS and information leakage attacks. Finally, we discussed the related work in this domain and laid down the challenges ahead.

REFERENCES

[1] Accelize. https://www.accelize.com/.
[2] Adaptable. intelligent. https://www.xilinx.com/.
[3] Amazon web services. https://aws.amazon.com/ec2/instance-types/f1/.
[4] Data center solutions, iot, and pc innovation. https://www.intel.com/.
[5] Global high performance computing market-trends forecast, 2017-2023. https://www.wiseguyreports.com/sample-request/3403732-global-high-performance-computing-market-trends-forecast-2017-2023.
[6] Intel fpga opencl sdk. https://fpgasoftware.intel.com/opencl/19.1/?edition=pro&download_manager=direct.
[7] Intel quartus prime software suite. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.
[8] Intel-sa-00284. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00284.html.
[9] Intel-sa-00311. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00311.html.
[10] Intel® stratix® 10 fpgas overview - high performance stratix® fpga. https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html.
[11] Nimbix. https://www.nimbix.net/.
[12] Pci sig specifications. https://pcisig.com/specifications.
[13] Tacc. 2018. catapult - texas advanced computing center. https://www.tacc.utexas.edu/systems/catapult.
[14] Terasic de10-pro fpga accelerator board. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=13&No=1144&PartNo=1.
[15] Vivado design suite user guide - partial reconfiguration. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf.
[16] Fpga configuration via protocol, May 2011. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01132-stxv-cvpcie.pdf.
[17] Field programmable gate array (fpga) market research report- forecast 2023. Technical report, 2019.

[18] Intel fpga rte for opencl pro edition: Getting started guide. *Programming Guide*, 2019.

[19] Intel high level synthesis compiler: Reference manual. *Reference Manual*, 2019.

[20] Intel quartus prime pro edition user guide: Partial reconfiguration. *UG-20136*, September 2019.

[21] Intel® fpga sdk for opencl™ pro edition. *Programming Guide*, April 2019.

[22] Rodrigo Branco and Shay Gueron. Blinded random corruption attacks. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 85–90. IEEE, 2016.

[23] Ravi Budruk, Don Anderson, and Tom Shanley. *PCI express system architecture.* Addison-Wesley Professional, 2004.

[24] Sumanta Chaudhuri. A Security Vulnerability Analysis of SoCFPGA Architectures. *Proceedings of the 55th Annual Design Automation Conference on - DAC '18*, pages 1–6, 2018.

[25] David Chisnall. *The definitive guide to the xen hypervisor.* Pearson Education, 2008.

[26] Jake Edge. Dma and get_user_pages(), 2018. https://lwn.net/Articles/774411/.

[27] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 57–72. ACM, 2001.

[28] Tom Feist. Vivado design suite. *White Paper*, 5:30, 2012.

[29] U Frisk. Pcileech: Direct memory access attack software.

[30] Ilias Giechaskiel, Kasper B Rasmussen, and Ken Eguro. Leaky wires: Information leakage and covert communication between fpga long wires. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 15–27. ACM, 2018.

[31] Dennis R.E. Gnad, Fabian Oboril, and Mehdi B. Tahoori. Voltage drop-based fault attacks on FPGAs using valid bitstreams. In *2017 27th International Conference on Field Programmable Logic and Applications, FPL 2017*, 2017.

[32] Allen D Householder, Garret Wassermann, Art Manion, and Chris King. The cert guide to coordinated vulnerability disclosure. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA PITTSBURGH United States, 2017.

[33] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–14. ACM, 2007.

[34] Taras Iakymchuk, Maciej Nikodem, and Krzysztof Kępa. Temperature-based covert channel in fpga systems. In *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–7. IEEE, 2011.

[35] Zheming Jin, Kazutomo Yoshii, Hal Finkel, and Franck Cappello. Evaluation of the opencl aes kernel using the intel fpga sdk for opencl. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[36] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[37] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[38] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[39] Jonas Krautter, Dennis R E Gnad, and Mehdi B Tahoori. FPGAhammer : Remote Voltage Fault Attacks on Shared FPGAs , suitable for DFA on AES. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):44–68, 2018.

[40] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *2014 IEEE Symposium on Security and Privacy*, pages 19–33. IEEE, 2014.

[41] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[42] Robert Love. *Linux system programming: talking directly to the kernel and C library.* " O'Reilly Media, Inc.", 2013.

[43] Anil Madhavapeddy and Satnam Singh. Reconfigurable data processing for clouds. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 141–145. IEEE, 2011.

[44] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Confidentiality issues on a gpu in a virtualized environment. In *International Conference on Financial Cryptography and Data Security*, pages 119–135. Springer, 2014.

[45] Andrea Miele. Buffer overflow vulnerabilities in cuda: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques*, 12(2):113–120, 2016.

[46] Benoît Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. Bypassing iommu protection against i/o attacks. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 145–150. IEEE, 2016.

[47] Ali Muhtaroglu, Greg Taylor, and Tawfik Rahal-Arabi. On-die droop detector for analog sensing of power supply noise. *IEEE Journal of solid-state circuits*, 39(4):651–660, 2004.

[48] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. Cuda leaks: a detailed hack for cuda and a (partial) fix. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):15, 2016.

[49] Christian Pilato, Kanad Basu, Francesco Regazzoni, and Ramesh Karri. Black-hat high-level synthesis: Myth or reality? *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(4):913–926, 2018.

[50] Christian Pilato, Kaijie Wu, Siddharth Garg, Ramesh Karri, and Francesco Regazzoni. Tainthls: High-level synthesis for dynamic information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):798–808, 2018.

[51] Nitin Pundir, Fahim Rahman, Mark Tehranipoor, and Farimah Farahmandi. Analyzing security vulnerabilities induced by high-level synthesis in socs, 2020.

[52] Falk Schellenberg, Dennis R.E. Gnad, Amir Moradi, and Mehdi B. Tahoori. An inside job: Remote power analysis attacks on FPGAs. In *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*, volume 2018-Janua, pages 1111–1116, 2018.

[53] Amazon Web Services. Aws ec2 fpga hardware and software development kit. https://github.com/aws/aws-fpga/, 2016.

[54] Shanquan Tian and Jakub Szefer. Temporal thermal covert channels in cloud fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 298–303. ACM, 2019.

[55] Julien Tinnes and Tavis Ormandy. Bypassing linux null pointer dereference exploit prevention (mmap_min_addr), 2009.

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

[57] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[58] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018.

[59] Ning-Yi Xu, Xiong-Fei Cai, Rui Gao, Lei Zhang, and Feng-Hsiung Hsu. Fpga acceleration of rankboost in web search engines. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(4):19, 2009.

[60] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, 2018.

[61] Jose Fernando Zazo, Sergio Lopez-Buedo, Yury Audzevich, and Andrew W Moore. A pcie dma engine to support the virtualization of 40 gbps fpga-accelerated network appliances. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2015.

[62] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. The feniks fpga operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7, 2017.

[63] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *ACM SIGPLAN Notices*, volume 49, pages 687–700. ACM, 2014.

[64] Kenneth M. Zick, Meeta Srivastav, Wei Zhang, and Matthew French. Sensing nanosecond-scale voltage attacks and natural transients in FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '13*, page 101, New York, New York, USA, 2013. ACM Press.