Memory Scraping Attack on Xilinx FPGAs: Private Data Extraction from Terminated Processes

Bharadwaj Madabhushi, Sandip Kundu, Daniel Holcomb
Department of Electrical and Computer Engineering
University of Massachusetts Amherst
{*bmadabhushi, kundu, dholcomb*}@umass.edu

***Abstract*—FPGA-based hardware accelerators are becoming increasingly popular due to their versatility, customizability, energy efficiency, constant latency, and scalability. FPGAs can be tailored to specific algorithms, enabling efficient hardware implementations that effectively leverage algorithm parallelism. This can lead to significant performance improvements over CPUs and GPUs, particularly for highly parallel applications. For example, a recent study found that Stratix 10 FPGAs can achieve up to 90% of the performance of a TitanX Pascal GPU while consuming less than 50% of the power. This makes FPGAs an attractive choice for accelerating machine learning (ML) workloads. However, our research finds privacy and security vulnerabilities in existing Xilinx FPGA-based hardware acceleration solutions. These vulnerabilities arise from the lack of memory initialization and insufficient process isolation, which creates potential avenues for unauthorized access to private data used by processes. To illustrate this issue, we conducted experiments using a Xilinx ZCU104 board running the PetaLinux tool from Xilinx. We found that PetaLinux does not effectively clear memory locations associated with a terminated process, leaving them vulnerable to memory scraping attack (MSA). This paper makes two main contributions. The first contribution is an attack methodology of using the Xilinx debugger from a different user space. We find that we are able to access process IDs, virtual address spaces, and pagemaps of one user from a different user space because of lack of adequate process isolation. The second contribution is a methodology for characterizing terminated processes and accessing their private data. We illustrate this on Xilinx ML application library.**

***Index Terms*—FPGA, PetaLinux, FPGA debugger, Memory Scraping Attack (MSA), Memory Residue**

I. Introduction

FPGAs have become increasingly popular as hardware accelerators in heterogeneous computing systems, especially in host-based environments such as data centers [1] and cloud computing systems [2]. Their adoption is driven by their reconfigurable nature [3], lower power consumption [4], high performance [5], scalability [6], low cost [7] and their ability to offload computationally intensive tasks from host CPUs [8], thereby reducing the overall load on the host CPUs whilst providing better quality of service to the customers. Figure 1 provides a general overview of the host-based system.

Multiple semiconductor companies are actively developing high-performance FPGAs to meet growing demand in heterogeneous computing systems, notably in data centers and for cloud service providers. AMD's Xilinx introduces Adaptive SOC FPGA [9] and Zynq system-on-a-chip (SoC). Microsoft's Project Brainwave leverages FPGAs for real-time AI inference acceleration [10], prioritizing low latency and high performance in both cloud and edge environments. IBM incorporates cloud-FPGA [11] solutions into its infrastructure as a service (IaaS), focusing on neural network modeling. Leading cloud providers, including AWS with Amazon EC2 F1 instances [12], leverage Xilinx and Altera FPGAs to offer FPGA acceleration services [13] commonly referring it as "FPGA-as-a-Service" (FaaS) or "Acceleration-as-a-Service" (AaaS) [14].

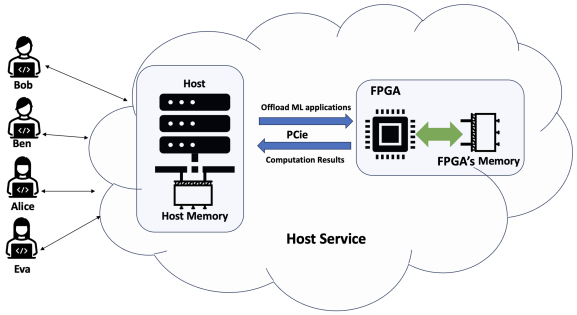

Fig. 1. A general host-based system.

The integration of an FPGA into a host-based system introduces security concerns described earlier in [14]. However, the following risks were not considered earlier.

A. Security Risks with Using Local Memory

When offloading compute intensive tasks from host processing unit (PU) to FPGA for acceleration, the FPGA's local DRAM is used to temporarily store [15] and reuse data before returning results to the host. However, this poses a security risk, as a subsequent guest accessing the FPGA DRAM after the first guest's process has ended may be able to retrieve memory residue from the first user. We demonstrate this in this paper by scraping memory residue in FPGA DRAM from a terminated computer vision machine learning application.

In CPU, a memory management unit (MMU) enforces memory isolation between multiple processes [16]. Similarly, in Xilinx FPGAs, a hypervisor like Xen manages isolation between multiple processes running on the FPGA [17]. How-

This research was funded in part by a grant from the National Science Foundation.

979-8-3503-4859-0/DATE24/© 2024 EDAA

ever, in CPUs, page tables are only accessible to the operating system (OS), not to any user process, including any program debugger a user may be running. We find that, unlike in CPUs, a Xilinx debugger has access to memory page tables. This is because Xen is not managed by the host OS, but rather configured by the user using PetaLinux (see Section I-C). We find this to be a gaping security hole that affects not only Xilinx FPGAs but also other embedded systems. For the purpose of this paper, we limit our focus to FPGAs.

Local Memory Scraping as an Attack Vector: In this work, we show that (*i*) Xilinx FPGAs do not perform automatic memory sanitization leaving memory residue, (*ii*) Xilinx debugger can be invoked from a second user space (even for a single tenant FPGA), and (*iii*) page tables are accessible from the debugger. We present a memory scraping technique that uses the above exploits to show how sensitive information about previous programs can be reconstructed.

Main Scientific Finding: Our main scientific finding is that many accelerators, including FPGAs, GPUs and various embedded systems, have their own private memory that is not under direct host OS control. For performance reasons, it is not practical to have the host OS mediate every local memory access. This creates a general security problem, and in this paper, we demonstrate a targeted attack on Xilinx FPGAs to highlight this problem.

B. Related Work

Zhou *et al*., [18] and Maurice *et al*., [19] previously attacked NVIDIA's heterogeneous memory systems in cloud-based systems, allowing adversaries to access and reconstruct data from terminated processes. A similar issue affects ARM Mali GPUs, where a new process can access a terminated process's pages due to inadvertent reuse of freed pages [20].

Memory Initialization Solutions: A number of studies have investigated rapid DRAM initialization techniques. Seshadri *et al*., proposed *RowClone*, a technique for initializing contiguous DRAM sections with zeros [21]. Seol *et al*., proposed a *RowReset*, a hardware-efficient memory initialization solution that manipulates VDD and VSS to DRAM banks [22]. However, these methods are best suited for continuous memory locations. In virtual environments, Address Space Layout Randomization (ASLR) [23] is employed to enhance security by randomizing DRAM memory locations, providing defense against memory corruption attacks. However, in multi-tenant FPGA settings with non-contiguous memory addresses [24], the aforementioned memory initialization solutions may inadvertently erase active guest user data. These solutions typically clear continuous memory locations when initializing memory for a terminated user, which can include active guest user data. Therefore, there is a need for more efficient solutions to initialize non-contiguous memory locations without jeopardizing active user data in local DRAM.

C. Target Platform

Leading cloud service providers, such as Amazon EC2 F1 [25] and Alibaba Cloud [26], have adopted AMD's Virtex Ultrascale+ FPGA family. Baidu's Apollo platform employs Zynq Ultrascale+ MPSoC FPGAs for self-driving vehicles [27]. All Ultrascale+ FPGAs [28] have onboard (local) memory for use by offloaded processes onto the FPGA board.

For our attack demonstration, we target the Zynq Ultrascale+ MPSoC ZCU104 board because it is more affordable than high-end Ultrascale+ Virtex boards. The ZCU104 has an architectural design similar to other Ultrascale+ FPGAs, making our attack scenario credible and relevant. The ZCU104 incorporates essential components like quad-core ARM Cortex-A53 APU, dual-core Cortex-R5 RPU, Mali-400 MP2 GPU, a high-definition video codec, and programmable logic component fabricated using 16nm FinFET+ technology. For generalizability studies we have reverified the attack on Zynq Ultrascale+ MPSoC ZCU102 board.

PetaLinux is the chosen software platform for ZCU104's system development [29]. It offers tools like command-line interfaces, generators, templates, and system configuration tools, including the Xen hypervisor as a selectable component [30]. PetaLinux's shell manager oversees hardware and software components for managing and functionality. Figure 2 provides an architectural overview with PetaLinux running on the APU.

D. Contributions

The main contributions of this paper are as follows:

1) We identify a security risk with using local memory.
2) We present a novel attack methodology that uses the Xilinx system debugger to mount a *system-channel attack*. A debugger is typically used to inspect the values of variables and optimize code by identifying performance bottlenecks and memory leaks. It can also track and monitor specific memory locations. We exploit this feature, coupled with the fact that local memory is not controlled by the host OS, bypassing access control to mount our attack. Our attack works on both single-tenant and multi-tenant FPGAs.
3) We use our novel attack methodology to scrape memory of a terminated process.
4) We present a novel data analysis technique involving offline profiling to learn high-value memory locations, then reading data from these locations in the scraped memory to reconstruct information about the target process.
5) We demonstrate our data analysis technique on Xilinx machine-learning model library identifying specific models used in terminated processes and revealing sensitive information such as input images and weights.

II. Adversary Model

This section outlines the adversary's goal, privileges, and capabilities.

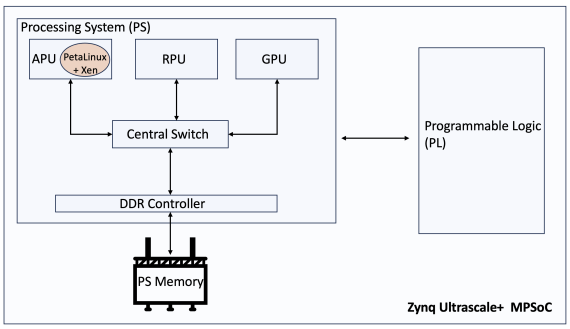

Fig. 2. A high level block diagram of Zynq Ultrascale+ MPSoC.

Adversary's goal: The adversary's goal is to (i) access data from local memory used previously by a terminated process and (ii) to perform analysis to reconstruct information from the terminated program. The adversary uses this information for compromising the privacy and security of the previous process entity.

Adversary's privileges: Adversaries can use a system debugger provided by the manufacturer, resulting in having unrestricted access to critical process details, including process IDs (pids), virtual address spaces, and pagemaps. Normally, such privileges are not available under a CPU OS. However, Xilinx System Debugger permits these privileges from user space giving adversaries access to data stored in the FPGA's DRAM at physical locations associated with a specific process (pid).

Adversary's access: The adversary has access to FPGA libraries and IPs used provided by Xilinx that are used by the user. By profiling the FPGA libraries and IPs provided by Xilinx, the adversary can, for instance, gain an understanding of the physical memory layout of machine learning models running on the board. This allows the adversary to identify where critical data, such as weights, vectors, and images, are stored. During the attack, the adversary uses this profiling data to identify the model and locate critical data in the FPGA's DRAM. The adversary can then attempt to reconstruct the associated image, compromising the previous user's data confidentiality.

III. Proposed Attack Methodology

The adversary follows a four-step sequence to extract and analyze data from FPGA's DRAM, enabling them to identify the executed model and potentially reconstruct the image, highlighting their capabilities outlined in II. The adversary's five steps are as follows:

1) **Polling for pid:** The adversary continuously monitors the system to identify the relevant process of interest, utilizing commands like "ps -ef" in Unix to extract the process ID (PID) associated with the targeted execution.

2) **Fetching virtual addresses and converting them to physical addresses:** Using the process ID, the adversary retrieves the virtual address locations of the targeted process from the heap mapping in the associated maps file. They then convert these virtual addresses into corresponding physical addresses within the FPGA's DRAM using information from the process's specific pagemaps file.

3) **Data extraction from physical addresses:** Once the targeted process is terminated or disconnected, the adversary proceeds to access and read the contents of the previously derived physical address locations within the FPGA's DRAM. By doing so, they gain access to the data stored by the terminated process.

4) **Analysis of extracted data:** Once the data is extracted the adversary now proceeds to analyze the data.
   a) ***Identifying models from strings:*** The adversary analyzes the FPGA DRAM data for distinct patterns or signatures of different models. Using criteria like keywords or known model names (e.g. "resnet50", "squeezenet"), they identify the model run by the targeted process based on the presence of similarly named libraries and data structures in memory.
   b) ***Reconstructing image:*** Depending on the model and whether it accepts an image as input, the adversary might attempt to reconstruct this input image. This is achievable due to the adversary's possession of knowledge about the physical memory layout of the identified model, which was acquired through offline profiling. Leveraging this information, the adversary can pinpoint the exact location where the image is stored and make an attempt at reconstruction.

IV. Experimental Setup

Setting up Target board: To conduct the experiments with the Xilinx ZCU104 FPGA board, we followed Xilinx's step-by-step instructions outlined in [31]. Figure 3 illustrates the target board.

1) The Xilinx-provided OS image for the ZCU104 board is flashed to an SD card. This image contains the Petalinux embedded OS and necessary software tools. The SD card

is then inserted into the ZCU104 board and powered on to boot the system.

2) After booting the board, we established a remote connection via the Ethernet interface to communicate and interact with it.

3) Finally, we installed the Vitis AI runtime on the target board, which provides various pre-built machine learning models from different vendors for testing and experimentation.

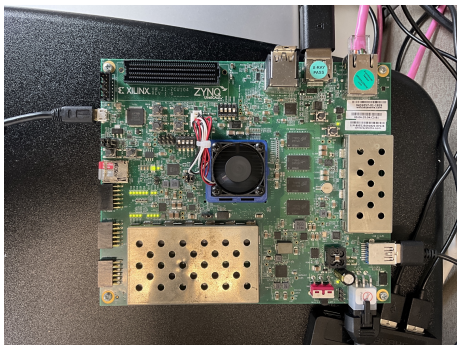

Fig. 3. Target Board (Xilinx's Zynq ZCU104)

Victim model: The selected victim model for our experimentation is "resnet50_pt" (RESidual NETwork using PyTorch Framework), sourced from Xilinx's examples. We chose this model because of its widespread use in image recognition tasks. Additionally, Xilinx has supplied a dedicated image designed for use with the resnet50_pt model, which we employed in our experiments.

Corrupting the image: We intentionally corrupted the example image by replacing its pixel values with 0xFFFFFF. When the adversary reads data from the FPGA's DRAM and encounters a sequence of 0xFFFFFF values, it signifies the corrupted image used as input for the resnet50_pt model (as shown in Figure 4). This indicates that the corrupted image was not cleared from the FPGA's DRAM after the process terminated, underscoring the absence of proper memory management.

Implementing steps described in Section III: The attack uses two terminals: one for the attacker and one for the victim. The victim runs the resnet50_pt model while the attacker runs Steps 1 and 2. After the victim's process ID disappears, confirming it has ended, the attacker proceeds with Steps 3 and 4 in the attacker terminal to read the data from the FPGA's DRAM and identify the executed model and try to reconstruct images. Our code written in python automates the full attack process.

V. Results

In this section, we provide results with illustrations from each step described in Section III. These results also illustrate the implementation aspects of the attack.

Step 1. Polling for pids: Figures 5 and 6 illustrates the running processes (pids) obtained from the attacker's terminal by executing the ps -ef command. Figure 5 shows the processes before running the resnet50_pt model, while Figure 6 shows the processes after its execution.

(a) Original image

(b) Computed image

Fig. 4. The top image (a) represents an example input for the resnet50_pt model, provided by Xilinx. The bottom image (b) shows a corrupted version achieved by altering specific pixel locations within the original image. About 20% of the image has been intentionally omitted to highlight the original image is modified.

```
1389       2  0 03:51 ?        00:00:00 [kworker/3:0-events]
1390     843  0 03:52 pts/0    00:00:00 ps -ef
```

Fig. 5. (Step 1) Process list before victim model was run.

```
1389       2  0 03:51 ?        00:00:00 [kworker/3:0-events]
1391    2430 18 12:33 pts/1    00:00:00 ./resnet50_pt
                                /usr/share/vitis_ai_library/models/resnet50_pt/resnet50_pt.xmodel
                                ../images/001.jpg
1392    1875  0 12:33 pts/0    00:00:00 ps -ef
```

Fig. 6. (Step 1) Process list after Victim model was run. Victim's pid is observed to be 1391.

Step 2. Fetching virtual addresses and converting them to physical addresses: We access the process's memory map using the command vim /proc/1391/maps for PID 1391, revealing the virtual address range of the heap, from 0xaaaaee775000 to 0xaaaaefd8a000 as shown in Figure 7. To convert these heap virtual addresses to physical addresses in the FPGA's DRAM, we have created C code based on our

offline training knowledge, as explained in Section II. This code maps the virtual addresses to a physical range, visible in Figure 8, from 0x61c6d730 to 0x61ec5e220.

```
aaaaee775000-aaaaefd8a000 rw-p 00000000 00:00 0                          [heap]
ffffb13b5000-ffffb6c1f000 rw-p 00000000 00:00 0          /dev/dri/renderD128
```

Fig. 7. (Step 2) Virtual address of the target process ranges from 0xaaaaee775000 to 0xaaaaefd8a000 in the heap.

```
xilinx-zcu104-20222:~# ./virtual_to_physical.out 1391 0xaaaaee775000
0x61c6d730
xilinx-zcu104-20222:~# ./virtual_to_physical.out 1391 0xaaaaefd8a000
0x61ec5e220
```

Fig. 8. (Step 2) Physical address values of the virtual addresses.

Step 3. Data extraction from physical addresses: To continuously monitor the termination of the specified process ID (PID), we repeatedly execute step 2. If the PID has been terminated successfully, it will no longer appear in the list of running processes. Figure 9 illustrates the absence of the PID from the process running list after its termination.

```
1389       2  0 03:51 ?        00:00:00 [kworker/3:0-events]
1401    1875  0 12:33 pts/0    00:00:00 ps -ef
```

Fig. 9. (Step 3) The figure shows that the PID 1391 absent in the process running list after it was terminated

```
xilinx-zcu104-20222:~# devmem 0x61c6d730
0x00000000
xilinx-zcu104-20222:~# devmem 0x61ec5e220
0xF7F5F8FD
```

Fig. 10. (Step 3) The figure provides an example of how "devmem" is utilized to read the data.

We proceeded by executing the command "devmem ⟨physical_address⟩" to retrieve data from the physical address locations of the FPGA's DRAM obtained in Step 3. Figure 10 shows an example of how devmem command is used to read the data. However, since these steps are automated, the devmem command is executed for all the physical address locations specified in Step 2.

Step 4.a Analysis of extracted data (Identifying models from strings): After entire data is extracted in Step 4, we now format this data it into a file, arranging the data into rows of eight nibbles each. Subsequently, we create a hex dump of this file by running "hexdump" on it to inspect if any meaningful, readable words emerge. By analyzing the snippet displayed in Figure 11, we discern that the converted string representation of the hex data reveals the presence of the model name resnet50_pt in the data readout.

Step 4.b Analysis of extracted data (Reconstructing image): Once it is established that the executed model corresponds to "resnet50_pt," we proceed with image reconstruction by searching for the identifier "FFFF FFFF" in the hexdump log,

```
xilinx-zcu104-20222:~# grep "resnet50" 1391_hexdump.log
6c73 2f72 6573 6e65 7435 305f 7074 2f72  ls/resnet50_pt/r
6876 6973 696f 6e2f 7265 736e 6574 3530  hvision/resnet50
```

Fig. 11. (Step 4.a) The figure shows that the model resnet50_pt is found in the read out from the FPGA memory.

which signifies the corrupted image used by the identified model. Figure 12 demonstrates the hexdump file created in Step 4.a and the observation of the image identifier in this file. This highlights that the data associated with pid 1391 is not cleared from the DRAM even after termination.

In practical experiments, we varied the pixel values of the input image. To precisely locate the image's starting point within the hexdump, we conducted offline profiling by changing pixel values to "0x555555." We then ran the "resnet50_pt" model offline with this modified image, repeating Steps 1 to 3. By analyzing the hexadecimal dump, we found the offset between the first occurrence of "5555 5555" and the hexdump file's start, specifically at row number "646768." As we only modified the image, preserving the underlying model's integrity, the image's offset within the heap remained consistent for any image used with this model. Utilizing this profiled information, we successfully retrieved and reconstructed the victim's input image from the data at the identified offset position within the victim's heap file, which was previously saved. This streamlined process enabled the reconstruction of the input image, leveraging insights gained from profiling various existing models.

```
0000 0000 0000 0000 9102 0000 0000 0000  ................
8007 71f1 aaaa 0000 7012 71f1 aaaa 0000  ..q.....p.q.....
....
....
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  UUUUUUUUUUUUUUUU
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  UUUUUUUUUUUUUUUU
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  UUUUUUUUUUUUUUUU
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  UUUUUUUUUUUUUUUU
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  UUUUUUUUUUUUUUUU
....
....
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  UUUUUUUUUUUUUUUU
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  UUUUUUUUUUUUUUUU
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  UUUUUUUUUUUUUUUU
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  UUUUUUUUUUUUUUUU
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  UUUUUUUUUUUUUUUU
...
```

Fig. 12. (Step 4.b) The figure shows the occurrence of "FFFF FFFF" the identifier of corrupted image used as an input by the model.

VI. Conclusion and Future Work

In this paper, we identified a security gap with using local memory that is not under host OS control. Typically, FPGA (or similar accelerator) manufacturers like to manage their own local memory for performance and efficiency. However, they must also equip their users to debug their program giving them access to local memory content from the debugger. Since the debugger accesses the local accelerator memory without host OS mediation, it falls on the FPGA manufacturer to restrict debugger access privileges. In this context, we find that the PetaLinux tool used by Xilinx to manage an FPGA

has major security holes. First, it allows unrestricted access to the page map tables. This enables an attacker process to scrape memory from a terminated victim process. Second, when a process terminates, it does not sanitize the physical memory used by the terminated process. Thus, an attacker can access the memory pages used by a terminated process. Third, it does not use any kind of randomization in physical page layout. This allows an attacker to learn about input or output data offsets, simply by learning from running the same program with its own input data. PetaLinux is a Xilinx supported tool to manage its FPGA cards. We conducted experiments on the Xilinx ZCU104 board using the Xilinx SDK to execute a machine learning program (referred to as the victim process). Through a systematic and step-by-step approach, we successfully showcased how an attacker can gain access to memory pages from the process, allowing them to deduce the specific program that was running and discern the input used. To enhance reproducibility and enable further exploration, we have automated the attack process and plan to release our code on GitHub.

VII. Ethical Disclosure

In line with responsible disclosure and ethical practices within the computer security research community, we reported these findings to AMD/Xilinx on July 14, 2023, along with all relevant details. AMD, the parent company of Xilinx acknowledged the validity of the attack on August 23, 2023.

References

[1] C. Kachris, B. Falsafi, and D. Soudris, *Hardware Accelerators in Data Centers*. Springer, 2019.

[2] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized fpga accelerators for efficient cloud computing," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 430–435.

[3] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–10.

[4] A. HajiRassouliha, A. J. Taberner, M. P. Nash, and P. M. Nielsen, "Suitability of recent hardware accelerators (dsps, fpgas, and gpus) for computer vision and image processing algorithms," *Signal Processing: Image Communication*, vol. 68, pp. 101–119, 2018.

[5] S. Ullah, S. Rehman, M. Shafique, and A. Kumar, "High-performance accurate and approximate multipliers for fpga-based hardware accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 211–224, 2021.

[6] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Fpgas in the cloud: Booting virtualized hardware accelerators with openstack," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 109–116.

[7] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, 2014.

[8] L. Sommer, J. Korinth, and A. Koch, "Openmp device offloading to fpga accelerators," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2017, pp. 201–205.

[9] AMD, "Adaptive SOC," https://www.amd.com/en/newsroom/press-releases/2023-6-27-amd-introduces-world-s-largest-fpga-based-adaptive.html, 2023, [Online; accessed 13-September-2023].

[10] Microsoft, "Project Brainwave," https://www.microsoft.com/en-us/research/project/project-brainwave/, 2023, [Online; accessed 13-September-2023].

[11] IBM, "cloud FPGA," https://www.zurich.ibm.com/cci/cloudFPGA/, 2023, [Online; accessed 13-September-2023].

[12] Amazon, "EC2 F1," https://aws.amazon.com/ec2/instance-types/f1/, 2023, [Online; accessed 13-September-2023].

[13] M. Kawser Ahmed, J. Mandebi, S. K. Saha, and C. Bobda, "Multi-Tenant Cloud FPGA: A Survey on Security," *arXiv e-prints*, p. arXiv:2209.11158, Sep. 2022.

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

[15] C. Bobda, J. M. Mbongue, P. Chow, M. Ewais, N. Tarafdar, J. C. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser *et al.*, "The future of fpga acceleration in datacenters and the cloud," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 3, pp. 1–42, 2022.

[16] J. M. Mbongue, D. T. Kwadjo, A. Shuping, and C. Bobda, "Deploying multi-tenant fpgas within linux-based cloud infrastructure," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 2, pp. 1–31, 2021.

[17] Xilinx, "Use of Hypervisors," https://docs.xilinx.com/r/en-US/ug1137-zynq-ultrascale-mpsoc-swdev/Use-of-Hypervisors, 2023, [Online; accessed 13-September-2023].

[18] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang, and R. Liu, "Vulnerable gpu memory management: towards recovering raw data from gpu," *arXiv preprint arXiv:1605.06610*, 2016.

[19] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "Confidentiality issues on a gpu in a virtualized environment," in *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers 18*. Springer, 2014, pp. 119–135.

[20] GHSL, "Use of Hypervisors," https://securitylab.github.com/advisories/GHSL-2022-053_Arm_Mali/, 2023, [Online; accessed 13-September-2023].

[21] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch *et al.*, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 185–197.

[22] H. Seol, W. Shin, J. Jang, J. Choi, J. Suh, and L.-S. Kim, "In-dram data initialization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 11, pp. 3251–3254, 2017.

[23] H. Marco-Gisbert and I. Ripoll, "On the effectiveness of full-aslr on 64-bit linux," in *Proceedings of the In-Depth Security Conference*, 2014.

[24] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs." in *NDSS*, 2017.

[25] AMD, "AWS Ultrascale+," https://www.xilinx.com/support/university/aws-f1.html#:~:text=The%20Amazon%20Elastic%20Compute%20Cloud,7%2C%20on%2Ddemand%20environment., 2023, [Online; accessed 13-September-2023].

[26] Xilinx, "Alibaba Ultrascale+," https://www.xilinx.com/publications/powered-by-xilinx/xilinx-alibaba-case-study.pdf, 2023, [Online; accessed 13-September-2023].

[27] ElectronicDesign, "Baidu Ultrascale+," https://www.electronicdesign.com/markets/automotive/article/21119589/xilinx-soc-fpga-powers-baidus-apollo-driverless-platform, 2023, [Online; accessed 13-September-2023].

[28] AMD, "Ultrascale+ architectures," https://docs.xilinx.com/v/u/en-US/ds890-ultrascale-overview, 2023, [Online; accessed 13-September-2023].

[29] ——, "Petalinux," https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html, 2023, [Online; accessed 13-September-2023].

[30] ——, "Use-of-hypervisors," https://docs.xilinx.com/r/en-US/ug1137-zynq-ultrascale-mpsoc-swdev/Use-of-Hypervisors, 2023, [Online; accessed 13-September-2023].

[31] Xilinx, "Vitis AI User Guide (UG1414). Xilinx Documentation," https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/For-Edge-DPUCZDX8G/DPUCVDX8G, 2023, [Online; accessed 19-July-2023].