Resurrection Attack: Defeating Xilinx MPU's Memory Protection

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

2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) | 979-8-3503-7394-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/HOST55342.2024.10545396

***Abstract*—Memory protection units (MPUs) are hardware-assisted security features that are commonly used in embedded processors such as the ARM 940T, Infineon TC1775, and Xilinx Zynq. MPUs partition the memory statically, and set individual protection attributes for each partition. MPUs typically define two protection domains: user mode and supervisor mode. Normally, this is sufficient for protecting the kernel and applications. However, we have discovered a way to access a process memory due to a vulnerability in Xilinx MPU (XMPU) implementation that we call *Resurrection Attack*. We find that XMPU security policy protects user memory from unauthorized access when the user is active. However, when a user's session is terminated, the contents of the memory region of the terminated process are not cleared. An attacker can exploit this vulnerability by gaining access to the memory region after it has been reassigned. The attacker can read the data from the previous user's memory region, thereby compromising the confidentiality. To prevent the Resurrection Attack, the memory region of a terminated process must be cleared. However, this is not the case in the XMPU implementation, which allows our attack to succeed. The Resurrection Attack is a serious security flaw that could be exploited to steal sensitive data or gain unauthorized access to a system. It is important for users of Xilinx FPGAs to be aware of this vulnerability until this flaw is addressed.**

***Index Terms*—FPGA, Process Memory, Xilinx Memory Protection Unit (XMPU), Unauthorized Access, Memory Initialization**

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are versatile integrated circuits that can be programmed and reconfigured to perform any logic function, from simple logic operations to complex computations. FPGAs can be used to implement algorithms directly in hardware, which can often lead to greater performance and power efficiency. Consequently, FPGAs are increasingly being used in various applications, including networking, critical infrastructure, aerospace, defense, and finance. FPGAs are also being used in high-performance cloud computing systems such as Amazon EC2 F1 instances [1] due to their efficiency and programmability even during runtime. From a security perspective, the ability to dynamically reconfigure FPGAs, also known as runtime update support, can be a major security vulnerability.

There have been several studies on security of FPGAs [2], [3]. They primarily focus static attack vectors such as supply chain vulnerabilities [4], malicious logic insertion [5], Trojans [6], backdoor attacks on FPGAs including timing violation induced faults, replay attacks [7], [8], and bit-stream tampering [9].

There has also been studies on physical attacks on FPGAs. One type of physical attack on FPGAs is a remote power side-channel attack where an attacker can rent an FPGA instance, build power monitors, and use a power analysis attack to steal secret information [10]. Attackers can reverse engineer FPGA logic by analyzing the bitstream, a binary file containing the configuration data [11]–[14]. Fault-injection attacks aim to exploit vulnerabilities in the FPGA design by injecting faults into the system [15]–[17].

Other attacks target FPGA interface components. For example, Weissman *et al*., performed a row-hammer attack on CPU main memory from the FPGA [18] and Ye *et al*., exploited the new attack surface between CPU-FPGA system [19]. Tin *et al*., demonstrated how PCIe contention can be used to attack the security of FPGAs in cloud data centers. They showed that by identifying instances of PCIe contention among FPGA slots, they could accurately correlate co-located FPGAs and their corresponding instance allocations. This information could then be used to launch other attacks [20]. Giechaskiel *et al*., showed how PCIe contention could be used to establish covert and side channels for covert transmission of information between virtual machines (VMs) [21].

There has also been studies on software based attacks. For example, malwares running on CPU can access the Block RAM (BRAM) through Direct Memory Access (DMA), and a hardware Trojan in FPGA can leak or modify video output frames of the CPU memory [19].

A. Multi-tenant FPGA vulnerabilities

In Multi-tenant FPGAs, a single FPGA fabric is shared between multiple users by partial reconfiguration, thereby increasing the utilization of FPGA logic. However, the adoption of multi-tenancy introduces novel security susceptibilities. Despite isolation mechanisms to keep user logic instances separate, attackers can exploit shared electrical components and attack co-located applications. Giechaskiel *et al*., used the FPGA interconnect to leak information using the observation of long wires that carry logical 1 reduce propagation delay of unconnected wire [22]. Similarly, Gnand et al. utilized shared power distribution networks among tenants to construct a covert communication channel bridging logically isolated

This work has been supported in part by a grant from the National Science Foundation (NSF).

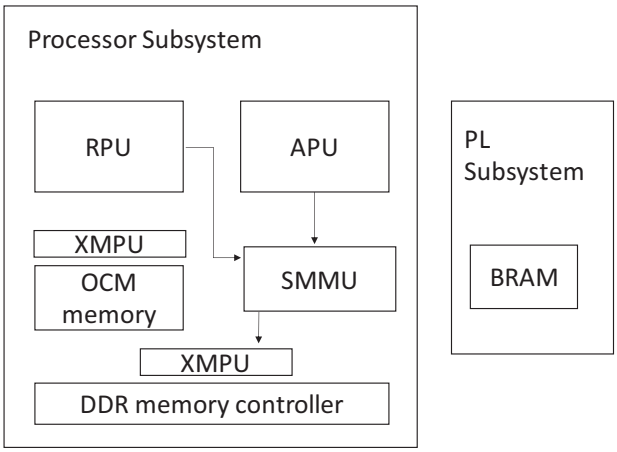

Fig. 1. Zynq UltraScale+ Architecture

users [23]. These studies underscore how spatially shared FPGA fabrics can be rendered vulnerable to the creation of side or covert channels through the exploitation of shared resources. In temporal sharing, FPGA fabric is shared by users in different time slots.

B. Memory protection in Xilinx MPSoC

In this section, we describe Xilinx Zynq UltraScale+ MPSoC hardware components that are used to isolate subsystems and protect them from each other. There are primarily two subsystems: the Processing System (PS) and the Programmable Logic (PL). And four memory regions double-data rate (DDR) memory, on-chip memory (OCM), tightly-coupled memory (TCM), and advanced eXtensible interface (AXI) block RAM in the PL system. It has eight XMPUs (Xilinx Memory Protection Units) to protect these memory regions from unauthorized access.

In this work, we focus on the Processing Subsystem. As shown in Figure 1, the PS has two main subsystems: the Application Processing Unit (APU), which is designed for general-purpose computing tasks, and the Real-time Processing Unit (RPU), which is designed for real-time applications. Xilinx protection unit is used to create memory isolation between processes. A subsytem can configured to run in protected or unprotected modes. XMPU authorizes whether a specific process is allowed to access a specific address. Out of the eight XMPUs, six of them are used to protect the DDR memory, generating an interrupt to notify any unauthorized access.

In addition to XMPUs, the Xilinx Zynq has a System Memory Management Unit (SMMU). SMMU extends the MMU capabilities of the processor core to the rest of the MPSoC architecture. SMMU translates virtual addresses to physical address space and can be used to restrict the reachable address space by creating multiple virtual address spaces, each with its security policy.

PetaLinux is an open-source software framework that can be used to develop and deploy embedded Linux systems on Xilinx FPGAs and SoCs.

With security blocks available Xilinx memory protection offers isolation by dividing the system memory into multiple regions, each with its own access permissions. This prevents unauthorized access to sensitive data and applications. Isolation can help to protect against unauthorized access, but it is not sufficient to secure a system. This is because memory residue from a terminated process can still be used to access sensitive data, even after the process has been isolated. Such memory may contain code, application data including sensitive information such as passwords and encryption keys. We call this attack a *Resurrection Attack*.

Related Attacks: Though, not the same, a closely related attack in the programming domain is known as Use-after-Free or UAF attack [24]. In UAF, a process, say a program written in C, allocates memory, frees it and reallocates it using a sequence of malloc(), free() and malloc(). The second malloc() inherits content from the first use without any sanitization. Similar attacks were mounted on GPUs by Lee *et al*., [25]. They presented methods for scraping GPU memory to reconstruct information about previously running programs, revealing private information.

Contributions: In this work, we show that (*i*) Xilinx FPGAs do not perform automatic memory sanitization leaving memory residue, (*ii*) XMPU protection mechanisms to isolate FPGA tasks or users can be altered in multiple ways to gain access to memory residue, and (*iii*) present a memory scraping technique to show how sensitive information about previous programs can be reconstructed.

II. Adversary Model

This section provides an overview of the adversary model, encompassing both the intent and capabilities of the adversary.

Background: The Xilinx MPU is a hardware-based memory protection unit that may be used to protect memory from unauthorized access. XMPU works by assigning each process a unique memory map, which specifies which memory addresses the process can access.

Attack question: When an active process is terminated, does Xilinx MPU clear the memory addresses that were allocated to it? Otherwise, if this memory is allocated to a new process without clearing it first, the new process will be able to gain unauthorized access to the data that was stored in these memory addresses previously, breaking a basic goal of memory protection between processes. This question is relevant because, Xilinx allows dynamic reconfiguration and clearing the memory would add to the reconfiguration time. So how does Xilinx make tradeoff between security and performance?

Assumptions about adversary's privileges and capabilities: Our target platform is the Xilinx Zynq ZCU102 board, featuring XMPU. TThe XMPU can be configured and controlled by secure masters, such as the Platform Management

Unit (PMU), Real-Time Processing Unit (RPU), or Application Processing Unit (APU) embedded in the FPGA.

The embedded operating system, PetaLinux, runs in a privileged mode that allows it to effectively program the XMPU registers through these masters. This programming capability enables PetaLinux to enforce memory isolation for specific memory addresses allocated to individual users.

In a multi-tenant scenario where multiple users share the FPGA concurrently, the need for memory protection is particularly important. Users in this environment can request memory protection from the XMPU to ensure the confidentiality and integrity of their allocated memory addresses.

However, once a user's session terminates, the memory addresses they were using may be available for reassignment to another user due to resource constraints. In such cases, the process of requesting memory protection and reprogramming the XMPU to provide protection for the new user's memory addresses is orchestrated through interactions with the secure masters, facilitated by PetaLinux.

The adversary in our study has the same privileges as any other user of the FPGA board. This means that they have the ability to request memory protection from the XMPU, use allocated memory addresses. However, the adversary's intent is different from that of regular users. The adversary is trying to exploit security vulnerabilities in the XMPU to gain unauthorized access to sensitive data of terminated process.

III. Proposed Attack Methodology

In this section we outline the steps need to be followed to show that memory is not initialized by a XMPU when a victim process is terminated.

Step 1. Polling process ID: In general, any application managed by an operating system (OS) is considered a process and is allocated a unique process ID (PID). The embedded OS on the FPGA also follows this approach when dealing with applications that are offloaded onto the FPGA. The embedded OS assigns a unique ID to the application, manages resource allocation, and oversees the entire lifecycle of the process.

The adversary monitors the victim's process ID (PID) by repeatedly polling it. This monitoring continues until the victim process, which was protected by the Xilinx Memory Protection Unit (XMPU), terminates. The adversary can poll the PID using the "`ps -ef`" command.

Step 2. XMPU protection request to alter XMPU settings: After the victim process terminates, the adversary requests memory allocation with protection from the embedded OS (PetaLinux). This changes the XMPU settings. However, the XMPU registers that specify the memory addresses remains unchanged. The adversary exploits this by simply requesting the activation of isolation, which gives the adversary access to the specified memory addresses.

Step 3. Scraping virtual addresses: At this point, the adversary does not know which physical address addresses are allocated to it by the PetaLinux system on the FPGA's onboard memory. To find out, the adversary uses the `ps -ef` command to retrieve its unique process ID. With this process ID, the adversary can then access the virtual address locations where its execution is taking place. This information can be obtained by inspecting the mapping of virtual addresses allocated to the process's heap. The details of this mapping are stored in the `maps` file associated with the process ID.

Step 4. Mapping virtual addresses to physical addresses: In this step, the adversary uses the page map file associated with the process ID to map the previously acquired virtual addresses to their corresponding physical addresses within the FPGA's onboard DRAM. It is important to note that this file is not accessible to users in traditional operating systems. However, the Xilinx debugger allows users to access map files. We exploit this loophole in the Xilinx debugger to obtain this information.

Step 5. Scraping memory residue: After obtaining the physical address, the adversary can read the content of the onboard DRAM of the FPGA. This step allows the adversary to scrape the memory residue from the terminated process. The memory residue can then be analyzed to determine what the previous process was doing and what data it was processing.

Step 6. Analysis of data scraped from victim: We illustrate how to analyze the data scraped from victim and gain access to information about the victim process using the following steps:

1) *Profiling*: The adversary first profiles the target victim processes to understand their memory layout. This involves identifying the relative memory addresses where critical data, such as signatures, vectors, or scores, is located. The adversary can use existing models or libraries in the relevant domain to perform this profiling, and the results can be saved for reference during live attacks.
2) *Reconstruction*: The adversary identifies the victim process by matching it to the reference file created during profiling. Once the match is confirmed, the adversary can access contents of specific memory locations where critical data is expected to reconstruct information about the previous process. The reconstruction step can be automated using software tools, which would make it even easier for the adversary to carry out the attack. However, automated reconstruction is beyond the scope of this paper.

These steps can be used to breach the confidentiality of data that has been left unsanitized in the memory.

IV. Experimental Setup

In this section, the configuration process for the target board is outlined which is specifically tailored to facilitate the execution of the chosen attack scenario. We delve into the details of configuring the XMPU to enable and disable isolation for memory addresses relevant to the attack. This involves replicating and adapting the steps provided in the reference document offered by Xilinx [26]. Furthermore, we

Fig. 2. Target Board (Xilinx's Zynq ZCU102)

present the specific registers that need to be programmed for the attack, ensuring comprehensive coverage of the setup process.

Setting up target board: We conducted our experiments on the Xilinx Zynq UltraScale+ MPSOC ZCU102 Z1 Rev1.1 FPGA board as shown in Figure 2 . This board is based on the Zynq UltraScale+ MPSoC, which is a multiprocessing system-on-chip that includes a dual-core ARM Cortex-A72 processor, a quad-core ARM Cortex-R5F processor, and a Mali-400 MP2 graphics processing unit. It also has 1GB of DDR4 SDRAM, 128MB of QSPI flash memory, and a variety of I/O ports. We followed the step-by-step instructions from Xilinx [26] to launch Peta-Linux. PetaLinux is responsible for tasks such as assigning process PIDs, allocating resources, scheduling, and terminating processes.

1) The process begins by flashing the OS image provided by Xilinx for the ZCU102 to an SD card. This image contains the operating system (PetaLinux) and software components necessary for the FPGA board. Once the OS image was successfully flashed to the SD card, we booted the board by inserting the SD card and turning it on.
2) Once the board is booted, we establish a remote connection to it using the Ethernet interface. This allows us to communicate and interact with the board from a remote location.
3) Finally, we install the Vitis AI runtime on the target board. This runtime environment includes a collection of example machine learning models from various vendors, providing a comprehensive set of pre-built models for testing and experimentation.

The above steps create a secure environment that allows us to configure XMPU security and explore its security vulnerabilities.

A. Attack sequence involving alteration of XMPU settings

The attack sequence for demonstrating lack of XMPU protection for a terminated process are derived from modifications to the steps outlined in the "APU Fault Injection Software Application Project" [27]. The original intent of this project is to showcase XMPU's protection capabilities for an active process. We will now provide an overview of attack sequence with the modified procedure below.

- **Victim Process**: The RPU runs an application on behalf of a process (victim), using specific memory addresses in the DRAM that are protected by the XMPU to prevent unauthorized access. The threat of unauthorized access comes from an adversary running on the APU. To demonstrate that the XMPU does not properly clear memory addresses designated for protection after a process finishes, we do the following:

  1) We create a APU application (proxy for victim) to write keywords into memory addresses designated for use by the RPU. These addresses were intended to be secure when the RPU accessed them.
  2) To run the application on RPU, we then asked PetaLinux to activate the XMPU's isolation mechanism. This entailed programming the XMPU with the relevant memory addresses and enabling isolation.
  3) Once activated, this application writes specific keywords to these memory locations. At this point, the adversary on the APU attempted to access the aforementioned memory addresses . The XMPU immediately prevented this access by generating an interrupt, thus upholding the security measures set by the XMPU.

- **Adversary Process**: We describe two scenarios for the attack process with and without XMPU protection.

  1) *XMPU disabled*: After termination of the victim process, we disable the XMPU protection for this memory addresses as the process is no longer alive. We then ran an adversary process to read the memory residue. We found that the adversary was able to read the keywords written by the victim process and the keywords written by the adversary, which were written before the RPU initiated a transaction. Even though the victim process had terminated and the XMPU was no longer active, the victim's data persisted within these memory locations allowing a different process to read it.
  2) *XMPU enabled*: In this scenario, when the victim's process is terminated, we request memory protection for executing the adversary process. The embedded OS allocated the same memory locations (as they remained unchanged in the XMPU registers) to the adversary process. Once again, our adversary process gained access to the victim's data and its own data (written prior to RPU's initiation), revealing a lack of proper memory initialization by the XMPU after a process has terminated.

Figures 3 to 7 illustrate the overview of attack procedure

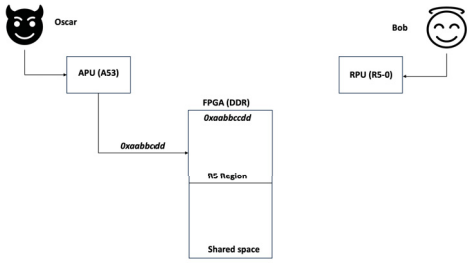

Fig. 3. Oscar writes into Bob's memory addresses when XMPU is disabled.

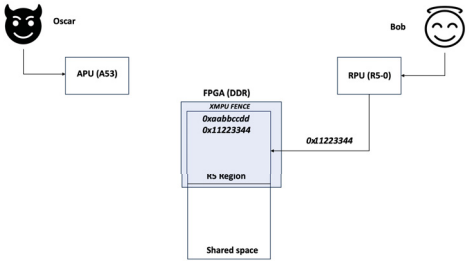

Fig. 4. Bob enables XMPU fence and writes his data.

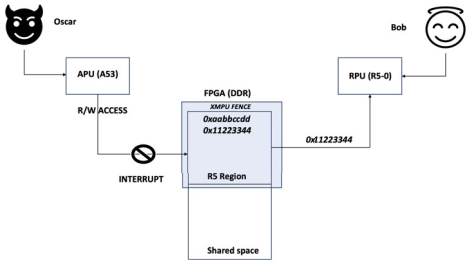

Fig. 5. Oscar's access attempt to Bob's addresses is denied, triggering an interrupt.

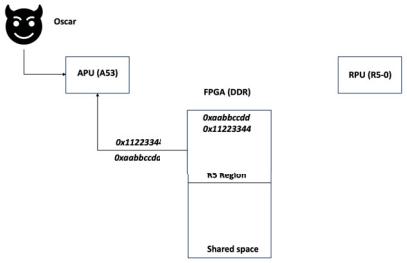

Fig. 6. After Bob's termination, the XMPU fence is lifted, granting Oscar access to Bob's memory contents.

described above. Oscar represents the adversary using the APU, while Bob represents the victim using the RPU. In Figure 3, Oscar exploits the disabled XMPU to write data into Bob's memory addresses. In Figure 4, Bob activates the XMPU fence and adds his data. In Figure 5, Oscar's attempt to access Bob's addresses is thwarted, resulting in an interrupt. In Figure 6, After Bob's process ends, the XMPU fence is lifted,

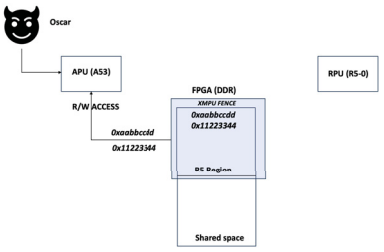

Fig. 7. Following Bob's termination, Oscar initiates a request for isolation from XMPU, thereby gaining access to the contents of Bob's memory.

allowing Oscar to access the contents within Bob's memory. In Figure 7, In an alternate scenario, the XMPU isolation remains active after Bob's termination. Oscar then initiates a request for isolation from the XMPU, which leads to the same memory addresses being accessed but with the ID in the XMPU configuration changed from Bob's to Oscar's.

The above sequence highlights the role of the XMPU in governing memory access when Bob's process is active, while also highlighting the potential exposure of Bob's data following his process termination.

B. Programming steps for executing the attack sequence

We illustrate the programming steps for demonstrating the lack of XMPU protection for a terminated process/application ran on the RPU. The RPU can access various types of memory, namely OCM, DDR, and ATCM. For the purpose of protection, memory regions are designated into three classes: secure, non-secure, or not-defined. The designations are explained below.

- *Secure (S):* This class of memory is accessible only to application running on RPU. The data stored in this memory is considered to be confidential and should not be accessible to any other process. These specific addresses are configured in the XMPU register with an ID that permits access only to the RPU application. Any attempt of unauthorized access triggers an error interrupt.
- *Non-Secure (NS):* This class of memory is accessible by both the task running on the RPU and tasks running on other IPs, such as the APU. The data stored in this memory is less critical and is intended to be shared among multiple IPs. In this scenario, the XMPU is configured with these memory addresses, but isolation is not enabled for them.
- *Not-Defined (ND):* The Not-defined class comprises memory addresses located within various memory regions that are currently unallocated and available for use by applications or IPs as needed. When a request is made for access to these addresses, their configuration in the XMPU can be determined based on the criticality of the user's data. They can be set as either Secure or Non-Secure addresses accordingly.

XMPU Register Configuration: The following XMPU registers are used to configure various memory regions as secure or non-secure:

- *Control registers*: These registers serve the purpose of enabling or disabling read and write access to the designated memory addresses as configured in the associated configuration registers. They are designed with a width of 32 bits, and their initial default value is set to 0x00000013. This default value implies that any user can access the memory regions without any isolation. To implement isolation and restrict access, these registers need to be programmed with the value 0x00000010.
- *SMID register*: The System Management ID (SMID) register is a 32-bit wide register, with the least significant 10 bits designated for configuration with the intended SMIDs required for accessing the respective addresses. If any other ID attempts to access these addresses, it triggers an interrupt, initiating an error handling process within the system. Additionally, the ID values can be masked with a 10-bit mask value to make them invisible, thereby preventing potential side-channel attacks on IDs.
- *END_HI/LOW and START_HI/LOW registers*: The combination of the START_HI and END_HI registers, when used together as a pair, constitute a 44-bit start address within a memory region. In this pair, START_HI uses its least significant 12 bits, while END_HI employs its entire 32 bits to create a 44-bit start address. Similarly, for the end address, it is formed by combining END_LOW[11:0] with END_HI[31:0]. The classification of these start and end addresses as either "secure" or "non-secure" is determined by the programming performed in the control registers, followed by the configuration of the corresponding ID that is permitted in the System Management ID (SMID) register.
- *LOCK registers*: When enabled (zeroth bit of the register), all XMPU registers can only be programmed once (during boot), and their settings can only be reset through an internal or external Power On Reset (POR). In scenarios involving multiple tenants, the lock register should be set to a value of 0x0. This allows for the flexible configuration of memory addresses within the XMPU registers in response to specific demands. It also facilitates the activation or deactivation of isolation as required.

XMPU registers are configured with memory addresses based on the criticality and necessity of data handled by a victim process, as well as the availability of addresses in the memory regions. These configurations classify memory regions as either "secure" or "non-secure" depending on their specific usage requirements. This flexible approach ensures that data security is tailored to the unique demands of each process while optimizing resource allocation.

Once the ZCU 102 board is successfully configured, we execute the attack sequence which allows us to investigate XMPU memory residue protection. We find that, Xilinx currently does not sanitize the memory after a process is terminated, which can leave memory residue with sensitive data.

```
Disabling XMPU
        Disabling ...
                Writing DDR_XMPU0_CTRL                  ...  PASS!
                Writing DDR_XMPU1_CTRL                  ...  PASS!
                Writing DDR_XMPU2_CTRL                  ...  PASS!
                Writing DDR_XMPU3_CTRL                  ...  PASS!
   Read/Write Of Various Memories
        Read/Write Of RPU(Secure) Memory
                Reading RPU_OCM_S_BASE                  ...  PASS!
                Writing RPU_OCM_S_BASE                  ...  PASS!
                Reading RPU_DDR_LOW_S_BASE              ...  PASS!
                Writing RPU_DDR_LOW_S_BASE              ...  PASS!
                Reading RPU_ATCM_S_BASE                 ...  PASS!
                Writing RPU_ATCM_S_BASE                 ...  PASS!
    APU has written 0x11223344 into RPU(Secure) Memory
```

Fig. 8. APU (adversary) writes data into secure memory addresses of RPU.

```
    Read/Write Of Various Memories
        Read/Write Of RPU(S) Memory
                Reading RPU_OCM_S_BASE                  ...  FAILED!
                Writing RPU_OCM_S_BASE                  ...  FAILED!
                Reading RPU_DDR_LOW_S_BASE              ...  FAILED!
                Writing RPU_DDR_LOW_S_BASE              ...  FAILED!
                Reading RPU_ATCM_S_BASE                 ...  FAILED!
                Writing RPU_ATCM_S_BASE                 ...  FAILED!
    APU is denied access into RPU(Secure) Memory
```

Fig. 9. APU (adversary) is denied access to the secure memory addresses of RPU.

V. Results

In this section, we provide results with illustrations from each step described in Section 3. These results also highlight the implementation aspects of the attack.

Watermarking of RPU memory region by APU: As described in the attack scenario, the APU, acting as a proxy for the victim, writes specified keyword 0x11223344 into specific memory locations of the RPU. These memory locations are then programmed into the specialized XMPU registers to create isolation during RPU usage. These APU write operations demonstrate that the XMPU does not initialize memory addresses for isolation before or after the start and end of a process, as described next.

Figure 8 shows how to disable isolation protections. First, the APU writes values to the XMPU's CTRL registers to disable isolation. These writes are valid transactions (denoted by "PASS!"), which prevents interrupts from being triggered by the error handler. Next, the APU writes data to different memories, including the On-chip Memory (OCM) used by secure masters, DDR, which is the on-board memory used widely across FPGA IPs, and Tightly Coupled Memory (ATCM). The RPU frequently accesses these memories from FPGA reset through shutdown.

XMPU protection activation and RPU watermarking: Then, the RPU process (victim) enables isolation protections and stores the keyword 0xaabbccdd in its memory addresses at locations that the APU had not written to previously. When the APU attempts to access these protected addresses, it is blocked and an interrupt is raised, which is handled by the

```
Disabling XMPU
        Enabling ...
                Writing DDR_XMPU0_CTRL                  ...  PASS!
                Writing DDR_XMPU1_CTRL                  ...  PASS!
                Writing DDR_XMPU2_CTRL                  ...  PASS!
                Writing DDR_XMPU3_CTRL                  ...  PASS!
  Read/Write Of Various Memories
        Read/Write Of RPU(Secure) Memory
                Reading RPU_OCM_S_BASE                  ...  PASS!
                Writing RPU_OCM_S_BASE                  ...  PASS!
                Reading RPU_DDR_LOW_S_BASE              ...  PASS!
                Writing RPU_DDR_LOW_S_BASE              ...  PASS!
                Reading RPU_ATCM_S_BASE                 ...  PASS!
                Writing RPU_ATCM_S_BASE                 ...  PASS!
    APU has read 0x11223344, 0xaabbccdd from RPU(Secure) Memory
```

Fig. 10. APU (Adversary) Accesses Both Its Data and the Victim's Data from RPU's Secure Memory addresses.

error handler. This demonstrates that the XMPU provides isolation for an active process.

Although the RPU enabling the XMPU and accessing its own addresses is similar to what is shown in Figure 8, Figure 9 shows that the APU is now denied access to the RPU's protected memory addresses.

XMPU protection deactivation and watermark check: After the victim process is terminated and XMPU isolation is deactivated, the APU can still access the memory locations where it originally wrote the value 0x11223344. It can also access the memory locations where the RPU had written the value "0xaabbccdd". This is shown in Figure 10. This shows that XMPU does not clear memory after a process terminates, and it does not clear memory even when isolation is disabled.

These experiments were conducted in a controlled environment, where we have full knowledge of the specific memory addresses that we wanted to manipulate. These addresses were all associated with the RPU subsystem. We also had the flexibility to program the XMPU, enabling or disabling it as needed for our attack scenario.

This simulated setup emulated a multi-tenant environment, which is a real-world scenario where adversarial applications may try to infiltrate the memory addresses of a victim application running on the RPU. We did not just focus on RPU memory, but also extended our analysis to include shared memory space between the APU and RPU, and various peripheral components.

Figure 11 illustrates our findings based on the ZCU102 platform. This figure shows that the XMPU can effectively isolate active processes. However, it also reveals a critical vulnerability: the inability to sanitize memory after a process terminates or even when isolation is reset (*i.e.*, transitioning from an enabled to a disabled state).

The results shown above indicate that an adversary can enter as a new process after XMPU's isolation is disabled, and then read the terminated victim's data. This suggests that XMPU does not sanitize memory after a process has been terminated, leaving it vulnerable to readout. The same results were observed when XMPU remained enabled as the adversary entered as a new process. In this case, instead of disabling and then re-enabling isolation for a new process, the XMPU configuration register was simply updated with a

```
---Starting Fault Injection Test (Running on the APU)---
   (S)=Secure, (NS)=Non-Secure, (ND)=Not-Defined
   Memories
        RPU_OCM_S_BASE              : OCM Secure Memory Base Address in RPU Sub-System
        RPU_OCM_NS_SHARED_BASE      : OCM Non-Secure Memory Base Address in both RPU and APU Sub-Systems
        RPU_ATCM_S_BASE             : R5 TCM Bank A Secure Memory Base Address in RPU Sub-System
        RPU_DDR_LOW_S_BASE          : DDR Secure Memory Base Address in RPU Sub-System
        RPU_DDR_LOW_NS_SHARED_BASE  : DDR Non-Secure Memory Base Address in both RPU and APU Sub-Systems
        APU_OCM_NS_SHARED_BASE      : OCM Non-Secure Memory Base Address in both APU and RPU Sub-Systems
        APU_DDR_LOW_NS_BASE         : DDR Non-Secure Memory Base Address in APU Sub-System
        APU_DDR_LOW_NS_SHARED_BASE  : DDR Non-Secure Memory Base Address in Both APU and RPU Sub-Systems
        UNDEFINED_DDR_MEMORY_BASE   : Memory Base Address Not Defined in Any Sub-System Peripherals
        APU_UART0_NS_START          : Non-Secure UART0 in APU Sub-System
        APU_SWDT0_NS_START          : Non-Secure UART0 in APU Sub-System
        APU_TTC0_NS_START           : Non-Secure UART0 in APU Sub-System
        APU_UART0_NS_START          : Non-Secure UART0 in APU Sub-System
        SHARED_GPIO_NS_START        : Shared Non-Secure GPIO
        RPU_UART1_S_START           : Secure UART1 in RPU Sub-System
        RPU_SWDT1_S_START           : Secure SWDT1 in RPU Sub-System
        RPU_TTC1_S_START            : Secure TTC1 in RPU Sub-System
        RPU_I2C1_S_START            : Secure I2C1 in RPU Sub-System
        DDR_XMPU0/1/2/3_CTRL        : Enables and Disables Security isolation
****************************************
 Disabling XMPU
        Disabling ...
                Writing DDR_XMPU0_CTRL                  ...  PASS!
                Writing DDR_XMPU1_CTRL                  ...  PASS!
                Writing DDR_XMPU2_CTRL                  ...  PASS!
                Writing DDR_XMPU3_CTRL                  ...  PASS!
 Read/Write Of Various Memories
        Read/Write Of RPU(S) Memory
                Reading RPU_OCM_S_BASE                  ...  PASS!
                Writing RPU_OCM_S_BASE                  ...  PASS!
                Reading RPU_DDR_LOW_S_BASE              ...  PASS!
                Writing RPU_DDR_LOW_S_BASE              ...  PASS!
                Reading RPU_ATCM_S_BASE                 ...  PASS!
                Writing RPU_ATCM_S_BASE                 ...  PASS!
        Read/Write Of RPU(NS) Memory
                Reading RPU_OCM_NS_SHARED_BASE          ...  PASS!
                Writing RPU_OCM_NS_SHARED_BASE          ...  PASS!
                Reading RPU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
                Writing RPU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
        Read/Write Of APU(S) Memory
               ---This combination does not exist
               ---APU has no (S)ecure memory
        Read/Write Of APU(NS)
                Reading APU_OCM_NS_SHARED_BASE          ...  PASS!
                Writing APU_OCM_NS_SHARED_BASE          ...  PASS!
                Reading APU_DDR_LOW_NS_BASE             ...  PASS!
                Writing APU_DDR_LOW_NS_BASE             ...  Skipped to avoid memory collision!
                Reading APU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
                Writing APU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
        Read/Write Of Undefined (ND) Memory
                Reading UNDEFINED_DDR_MEMORY_BASE       ...  PASS!
                Writing UNDEFINED_DDR_MEMORY_BASE       ...  PASS!
   Reading Sub-System Peripherals
        APU Peripherals
                Reading APU_UART0_NS_START              ...  PASS!
                Reading APU_SWDT0_NS_START              ...  PASS!
                Reading APU_TTC0_NS_START               ...  PASS!
        RPU Peripherals
                Reading RPU_UART1_S_START               ...  PASS!
                Reading RPU_SWDT1_S_START               ...  PASS!
                Reading RPU_TTC1_S_START                ...  PASS!
                Reading RPU_I2C1_S_START                ...  PASS!
        Shared Peripherals
                Reading SHARED_GPIO_NS_START            ...  PASS!
APU has written 0x11223344 into DRAM
****************************************
 APU tries to disable XMPU which is now programmed by RPU
        Disabling ...
                Writing DDR_XMPU0_CTRL                  ...  FAILED!
                Writing DDR_XMPU1_CTRL                  ...  FAILED!
                Writing DDR_XMPU2_CTRL                  ...  FAILED!
                Writing DDR_XMPU3_CTRL                  ...  FAILED!
   Read/Write Of Various Memories
        Read/Write Of RPU(S) Memory
                Reading RPU_OCM_S_BASE                  ...  FAILED!
                Writing RPU_OCM_S_BASE                  ...  FAILED!
                Reading RPU_DDR_LOW_S_BASE              ...  FAILED!
                Writing RPU_DDR_LOW_S_BASE              ...  FAILED!
                Reading RPU_ATCM_S_BASE                 ...  FAILED!
                Writing RPU_ATCM_S_BASE                 ...  FAILED!
        Read/Write Of RPU(NS) Memory
                Reading RPU_OCM_NS_SHARED_BASE          ...  PASS!
                Writing RPU_OCM_NS_SHARED_BASE          ...  PASS!
                Reading RPU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
                Writing RPU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
        Read/Write Of APU(S) Memory
               ---This combination does not exist
               ---APU has no (S)ecure memory
        Read/Write Of APU(NS)
                Reading APU_OCM_NS_SHARED_BASE          ...  PASS!
                Writing APU_OCM_NS_SHARED_BASE          ...  PASS!
                Reading APU_DDR_LOW_NS_BASE             ...  PASS!
                Writing APU_DDR_LOW_NS_BASE             ...  Skipped to avoid memory collision!
                Reading APU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
                Writing APU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
        Read/Write Of Undefined (ND) Memory
                Reading UNDEFINED_DDR_MEMORY_BASE       ...  PASS!
                Writing UNDEFINED_DDR_MEMORY_BASE       ...  PASS!
   Reading Sub-System Peripherals
        APU Peripherals
                Reading APU_UART0_NS_START              ...  PASS!
                Reading APU_SWDT0_NS_START              ...  PASS!
                Reading APU_TTC0_NS_START               ...  PASS!
        RPU Peripherals
                Reading RPU_UART1_S_START               ...  FAILED!
                Reading RPU_SWDT1_S_START               ...  FAILED!
                Reading RPU_TTC1_S_START                ...  FAILED!
                Reading RPU_I2C1_S_START                ...  FAILED!
        Shared Peripherals
                Reading SHARED_GPIO_NS_START            ...  PASS!
 APU cannot read  DRAM
****************************************
 APU disabling XMPU (Victim on RPU is terminated)
        Disabling ...
                Writing DDR_XMPU0_CTRL                  ...  PASS!
                Writing DDR_XMPU1_CTRL                  ...  PASS!
                Writing DDR_XMPU2_CTRL                  ...  PASS!
                Writing DDR_XMPU3_CTRL                  ...  PASS!
  Read/Write Of Various Memories
        Read/Write Of RPU(S) Memory
                Reading RPU_OCM_S_BASE                  ...  PASS!
                Writing RPU_OCM_S_BASE                  ...  PASS!
                Reading RPU_DDR_LOW_S_BASE              ...  PASS!
                Writing RPU_DDR_LOW_S_BASE              ...  PASS!
                Reading RPU_ATCM_S_BASE                 ...  PASS!
                Writing RPU_ATCM_S_BASE                 ...  PASS!
        Read/Write Of RPU(NS) Memory
                Reading RPU_OCM_NS_SHARED_BASE          ...  PASS!
                Writing RPU_OCM_NS_SHARED_BASE          ...  PASS!
                Reading RPU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
                Writing RPU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
        Read/Write Of APU(S) Memory
               ---This combination does not exist
               ---APU has no (S)ecure memory
        Read/Write Of APU(NS)
                Reading APU_OCM_NS_SHARED_BASE          ...  PASS!
                Writing APU_OCM_NS_SHARED_BASE          ...  PASS!
                Reading APU_DDR_LOW_NS_BASE             ...  PASS!
                Writing APU_DDR_LOW_NS_BASE             ...  Skipped to avoid memory collision!
                Reading APU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
                Writing APU_DDR_LOW_NS_SHARED_BASE      ...  PASS!
 Read/Write Of Undefined (ND) Memory
                Reading UNDEFINED_DDR_MEMORY_BASE       ...  PASS!
                Writing UNDEFINED_DDR_MEMORY_BASE       ...  PASS!
   Reading Sub-System Peripherals
        APU Peripherals
                Reading APU_UART0_NS_START              ...  PASS!
                Reading APU_SWDT0_NS_START              ...  PASS!
                Reading APU_TTC0_NS_START               ...  PASS!
        RPU Peripherals
                Reading RPU_UART1_S_START               ...  PASS!
                Reading RPU_SWDT1_S_START               ...  PASS!
                Reading RPU_TTC1_S_START                ...  PASS!
                Reading RPU_I2C1_S_START                ...  PASS!
        Shared Peripherals
                Reading SHARED_GPIO_NS_START            ...  PASS!
 APU has read 0x11223344 and 0xaabbccdd from DRAM
---Fault Injection Test Complete---
```

Fig. 11. Extensive Attack Scenario by APU on RPU.

```
1874      1  0 Jul12 ?        00:00:00 /usr/sbin/dropbear -i -r /etc/dropbear/dropbear_rsa_host_key -B
1875   1874  0 Jul12 pts/0    00:00:00 -sh
2056      2  0 Jul12 ?        00:00:00 [kworker/3:2-cgroup_destroy]
2429      1  0 05:11 ?        00:00:00 /usr/sbin/dropbear -i -r /etc/dropbear/dropbear_rsa_host_key -B
2430   2429  0 05:11 pts/1    00:00:00 -sh
2456      2  0 05:14 ?        00:00:00 [kworker/2:0H]
2457      2  0 05:14 ?        00:00:00 [DPUCZDX8G_2]
2458      2  0 05:14 ?        00:00:00 [DPUCZDX8G_1]
2776      2  0 11:39 ?        00:00:00 [kworker/u8:1-events_unbound]
2802      2  0 11:55 ?        00:00:00 [kworker/3:1H]
2803      2  0 11:55 ?        00:00:00 [kworker/1:2H]
2811      2  0 12:04 ?        00:00:00 [kworker/0:2-events_power_efficient]
2819      2  0 12:16 ?        00:00:00 [kworker/u8:0-events_unbound]
2828      2  0 12:27 ?        00:00:00 [kworker/0:1-mm_percpu_wq]
2829      2  0 12:29 ?        00:00:00 [kworker/u8:2-events_unbound]
2831    849  0 12:30 ?        00:00:00 systemd-userwork
2832    849  0 12:30 ?        00:00:00 systemd-userwork
2833    849  0 12:30 ?        00:00:00 systemd-userwork
2834      2  0 12:33 ?        00:00:00 [kworker/0:0-events]
2835   2430 18 12:33 pts/1    00:00:00 critical_application.py
2836   1875  0 12:33 pts/0    00:00:00 ps -ef
```

Fig. 12. (Step 1) Process list before Victim model was terminated. Victim's pid is observed to be 2835.

```
1874      1  0 Jul12 ?        00:00:00 /usr/sbin/dropbear -i -r /etc/dropbear/dropbear_rsa_host_key -B
1875   1874  0 Jul12 pts/0    00:00:00 -sh
2056      2  0 Jul12 ?        00:00:00 [kworker/3:2-cgroup_destroy]
2429      1  0 05:11 ?        00:00:00 /usr/sbin/dropbear -i -r /etc/dropbear/dropbear_rsa_host_key -B
2430   2429  0 05:11 pts/1    00:00:00 -sh
2456      2  0 05:14 ?        00:00:00 [kworker/2:0H]
2457      2  0 05:14 ?        00:00:00 [DPUCZDX8G_2]
2458      2  0 05:14 ?        00:00:00 [DPUCZDX8G_1]
2776      2  0 11:39 ?        00:00:00 [kworker/u8:1-events_unbound]
2802      2  0 11:55 ?        00:00:00 [kworker/3:1H]
2803      2  0 11:55 ?        00:00:00 [kworker/1:2H]
2811      2  0 12:04 ?        00:00:00 [kworker/0:2-events_power_efficient]
2819      2  0 12:16 ?        00:00:00 [kworker/u8:0-events_unbound]
2824      2  0 12:21 ?        00:00:00 [kworker/0:0-mm_percpu_wq]
2825    849  0 12:25 ?        00:00:00 systemd-userwork
2826    849  0 12:25 ?        00:00:00 systemd-userwork
2827    849  0 12:25 ?        00:00:00 systemd-userwork
2828      2  0 12:27 ?        00:00:00 [kworker/0:1-events]
2829      2  0 12:29 ?        00:00:00 [kworker/u8:2-events_unbound]
2830   1875  0 12:30 pts/0    00:00:00 ps -ef
```

Fig. 13. (Step 1) Process list after victim model is terminated.

new ID to allow access. However, the memory addresses and isolation settings remained unchanged in the XMPU registers. As a result, the adversary process was still able to read these previously accessed memory locations from the terminated process, highlighting that XMPU did not initialize memory, consistent with the results shown above.

Next, we will illustrate how the adversary can access the locations of the previous process and attempt to determine its nature. This will be demonstrated through a scenario in which the victim executes a critical application named critical_application.py which only writes 0xffffffff into memory, and we will show how the adversary can access this data and leverage this information. The adversary runs an application named adversary.py which only scrapes data from the physical addresses obtained in the steps below.

Step 1. Polling for process ID: Figures 12 and 13 show the active processes (pids) obtained from the attacker's terminal by executing the ps -ef command. Figure 12 shows the processes running critical_application.py, while Figure 13 shows the processes after it is terminated.

After the victim is terminated, the adversary requests for its memory protection by isolation (Step2) and start execution. Figure 14 shows that the adversary starts executing its own code with pid 2840.

Step 3. Scraping virtual addresses: In the second step of the process, the adversary requests initiation of memory protection by isolation. This changes the ID value in the XMPU configuration register from the victim's to the adversary's, while leaving the memory addresses in the XMPU configuration unchanged (retaining the victim's settings).

```
1874      1  0 Jul12 ?        00:00:00 /usr/sbin/dropbear -i -r /etc/dropbear/dropbear_rsa_host_key -B
1875   1874  0 Jul12 pts/0    00:00:00 -sh
2056      2  0 Jul12 ?        00:00:00 [kworker/3:2-cgroup_destroy]
2429      1  0 05:11 ?        00:00:00 /usr/sbin/dropbear -i -r /etc/dropbear/dropbear_rsa_host_key -B
2430   2429  0 05:11 pts/1    00:00:00 -sh
2456      2  0 05:14 ?        00:00:00 [kworker/2:0H]
2457      2  0 05:14 ?        00:00:00 [DPUCZDX8G_2]
2458      2  0 05:14 ?        00:00:00 [DPUCZDX8G_1]
2776      2  0 11:39 ?        00:00:00 [kworker/u8:1-events_unbound]
2802      2  0 11:55 ?        00:00:00 [kworker/3:1H]
2803      2  0 11:55 ?        00:00:00 [kworker/1:2H]
2811      2  0 12:04 ?        00:00:00 [kworker/0:2-events_power_efficient]
2819      2  0 12:16 ?        00:00:00 [kworker/u8:0-events_unbound]
2824      2  0 12:21 ?        00:00:00 [kworker/0:0-mm_percpu_wq]
2825    849  0 12:25 ?        00:00:00 systemd-userwork
2826    849  0 12:25 ?        00:00:00 systemd-userwork
2827    849  0 12:25 ?        00:00:00 systemd-userwork
2828      2  0 12:27 ?        00:00:00 [kworker/0:1-events]
2829      2  0 12:29 ?        00:00:00 [kworker/u8:2-events_unbound]
2830   1875  0 12:30 pts/0    00:00:00 ps -ef
2840   2430 18 12:33 pts/1    00:00:00 python3 adversary.py
```

Fig. 14. (Step 2) Process list after adversary start execution. Adversary id is seen to be 2840.

```
aaab0fcf1000-aaab11306000 rw-p 00000000 00:00 0                          [heap]
ffffa52a5000-ffffa6b2f000 rw-p 00000000 00:00 0
ffffa8035000-ffffa805b000 rw-s 00000000 00:05 733                        /dev/dri/renderD128
```

Fig. 15. (Step 3) Virtual address of the adversary process ranges from 0xaaab0fcf1000 to 0xaaab11306000 in the heap.

It is important to note that when a process requests isolation via the embedded OS, it is assigned virtual memory addresses, which is mapped to the physical addresses pre-configured in the XMPU registers. Consequently, physical addresses remain invariant, even though the virtual addresses may change. Every process that requests isolation uses the same physical addresses for its execution in the FPGA's onboard memory unless the physical addresses in the XMPU registers are reconfigured differently by a secure master. Since these registers are not accessible from the user space, the adversary does not know the physical addresses at this point.

However, the adversary can simply requests isolation and learn the virtual addresses assigned to it by PetaLinux. In Step 4, we describe how these virtual addresses can then be converted to a physical addresses. This conversion reveals the physical memory locations that the victim previously used, as they are the same physical addresses that were initially configured in the XMPU registers.

To obtain critical information about the virtual address space, adversary executes the command vim /proc/2840/maps, which enables adversary to access the memory map of the process with the PID 2840. Figure 15 provides adversary with valuable insight into the heap's virtual address range, specifically ranging from 0xaaab0fcf1000 to 0xaaab11306000. This range signifies the virtual address space used and allocated by the victim model for storing its data.

Step 4. Mapping virtual addresses to physical addresses: Adversary executes the command xxd to perform a hexdump of the heap memory associated with the specified process ID (PID). The resulting output is saved to a file for further analysis. By using the grep command and searching for

```
aaab0fd8ef10: 0000 0000 0000 0000 0000 0000 0000 0000  ................
aaab0fd8ef20: ffff ffff ffff ffff ffff ffff ffff ffff  ................
aaab0fd8ef30: ffff ffff ffff ffff ffff ffff ffff ffff  ................
aaab0fd8ef40: ffff ffff ffff ffff ffff ffff ffff ffff  ................
aaab0fd8ef50: ffff ffff ffff ffff ffff ffff ffff ffff  ................
aaab0fd8ef60: ffff ffff ffff ffff ffff ffff ffff ffff  ................
aaab0fd8ef70: ffff ffff ffff ffff ffff ffff ffff ffff  ................
...
...
...
aaab0fd8f0d0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
aaab0fd8f0e0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
aaab0fd8f0f0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
aaab0fd8f100: ffff ffff ffff ffff ffff ffff ffff ffff  ................
aaab0fd8f110: ffff ffff ffff ffff ffff ffff ffff fffe  ................
aaab0fd8f120: ffff fcff fffc ffff feff fffe fffe ffff  ................
aaab0fd8f130: feff fffd ffff feff ffff fff4 f7f5 f8fd  ................
```

Fig. 16. (Step 4) This figure illustrates a hexadecimal dump of the victim's data (ffff ffff) located in the virtual memory addresses ranging from 0xaaab0fd8ef20 to 0xaaab0fd8f130.

```
xilinx-zcu102-20222:~/READ# devmem 0x70c6df20
0xFFFFFFFF
xilinx-zcu102-20222:~/READ# devmem 0x70c6df24
0xFFFFFFFF
xilinx-zcu102-20222:~/READ# devmem 0x70c6df28
0xFFFFFFFF
xilinx-zcu102-20222:~/READ# devmem 0x70c6df2c
0xFFFFFFFF
...
...
...
xilinx-zcu102-20222:~/READ# devmem 0x70c6e130
0xFEFFFFFD
xilinx-zcu102-20222:~/READ# devmem 0x70c6e124
0xFFFFFEFF
xilinx-zcu102-20222:~/READ# devmem 0x70c6e128
0xFFFFFFF4
xilinx-zcu102-20222:~/READ# devmem 0x70c6df88
0xF7F5F8FD
```

Fig. 17. (Step 5) The figure shows the data read is 0xFFFFFF which identifies that the read out is of victim's.

the pixel values ffff ffff, which represent the data written by victim application (critical_application.py). Figure 16 shows the memory range encompassing the victims writes of 0xffffffff, spanning from memory addresses 0xaaab0fd8ef20 to 0xaaab0fd8f130. To translate these virtual memory addresses to physical addresses, we refer to the information obtained from the page maps file associated with the PID. Thus, adversary obtains a range of 0x70c6df20 to 0x70c6e130 for physical addresses.

Step 5. Scraping memory residue: To read data from physical addresses, adversary uses the command devmem ⟨physical_address⟩. Thus, adversary retrieves data from the physical addresses obtained in Step 4. In order to demonstrate that adversary was able to retrieve the victim's data 0xffffffff (corrupted image) from these physical locations, adversary ran the devmem command for each individual physical memory location, ranging from 0x70c6df20 to 0x70c6e130. The results of each devmem command execution are illustrated in Figure 17. However, since the steps are automated, the devmem command is executed for all the physical address locations specified in Step 4.

Step 6. Analysis of data scraped from victim: In this example, we as adversary have knowledge of what the victim was writing, allowing us to identify it in the hexdump during Step 4. From there, we converted those virtual addresses

```
aaaaf1676000: 0000 0000 0000 0000 9102 0000 0000 0000  ................
aaaaf1676010: 0700 0200 0200 0200 0100 0100 0700 0700  ................
aaaaf1676020: 0600 0400 0600 0600 0300 0400 0000 0700  ................
aaaaf1676030: 0000 0400 0200 0700 0600 0100 0000 0000  ................
aaaaf1676040: 0000 0100 0200 0200 0200 0300 0100 0300  ................
aaaaf1676050: 0500 0000 0300 0100 0500 0100 0200 0000  ................
aaaaf1676060: 0000 0100 0200 0000 0000 0300 0000 0000  ................
aaaaf1676070: 0000 0000 0000 0000 0000 0100 0000 0000  ................
aaaaf1676080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
aaaaf1676090: 8007 71f1 aaaa 0000 7012 71f1 aaaa 0000  ..q.....p.q.....
....
....
....
aaaaf1713e60: 0000 0000 0000 0000 0000 0000 0000 0000  ................
aaaaf1713e70: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713e80: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713e90: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713ea0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713eb0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713ec0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713ed0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713ee0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713ef0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713f00: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713f10: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713f20: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713f30: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713f40: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713f50: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713f60: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713f70: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713f80: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713f90: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713fa0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713fb0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713fc0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713fd0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713fe0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1713ff0: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1714000: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1714010: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1714020: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1714030: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1714040: 5555 5555 5555 5555 5555 5555 5555 5555  UUUUUUUUUUUUUUUU
aaaaf1714050: 5555 5555 5555 5555 5555 5555 5555 5654  UUUUUUUUUUUUUUVT
aaaaf1714060: 5556 5255 5652 5755 5457 5554 5754 5657  UVRUVRWUTWUTWTVW
```

Fig. 18. (Step 4) Experiment with victim writing 0x55555555 into memory.

to physical addresses and proceeded to Step 5 to scrape memory residue from these addresses. However, in a real-world scenario, the adversary may not have prior knowledge of the type of application the victim is running or the domain it is associated with. Therefore, the adversary needs to identify the domains for which this board is being used as hardware accelerators and work on obtaining reference files from those domains for effective comparison, as explained in Section 3 in Step 6 - *Profiling* and *Reconstruction*.

Following the steps described in Section 3, we have shown that XMPU does not sanitize memory residue left by a terminated victim in both scenarios described under *Adversary Process* in Section 4. If the adversary has prior knowledge of the victim applications, it can mount a potent attack on the memory residue to reconstruct the victim usage scenario.

Our experiments did not stop at the victim writing 0xffffffff; we also tested various other identifiers, such as 0x55555555, as shown in Figure 18, to validate the robustness of our approach. These additional tests reaffirmed the effectiveness of our approach in exposing the XMPU security vulnerabilities.

VI. CONCLUSION

The Xilinx Memory Protection Unit promises protection for active processes by isolating them from each other. However, as this paper shows, XMPU does not clear the memory for terminated processes, leaving it exposed to attacker processes even when the victim process ran under XMPU protection. It is worth noting that memory reconfiguration solely takes place during the First Stage Boot Loader (FSBL) phase, immediately following a power-on reset. Erasing memory upon process termination can lead to a performance overhead and increased reconfiguration time, which may explain Xilinx's decision not to do so. However, as this paper has shown, failing to clear memory leaves private data belonging to the terminated

process exposed to other processes. This is a serious security flaw in the XMPU architecture.

REFERENCES

[1] Amazon EC2 F1 Instances — aws.amazon.com. https://aws.amazon.com/ec2/instance-types/f1/. [Accessed 01-09-2023].

[2] Hoda Naghibijouybari, Esmaeil Mohammadian Koruyeh, and Nael Abu-Ghazaleh. Microarchitectural attacks in heterogeneous systems: A survey. *ACM Computing Surveys*, 55(7):1–40, 2022.

[3] Shaza Zeitouni, Ghada Dessouky, and Ahmad-Reza Sadeghi. Sok: On the security challenges and risks of multi-tenant fpgas in the cloud. *arXiv preprint arXiv:2009.13914*, 2020.

[4] Vladimir Rožić, Bohan Yang, Jo Vliegen, Nele Mentens, and Ingrid Verbauwhede. The monte carlo puf. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, 2017.

[5] Jonathan Cruz, Christopher Posada, Naren Vikram Raj Masna, Prabuddha Chakraborty, Pravin Gaikwad, and Swarup Bhunia. A framework for automated exploration of trojan attack space in fpga netlists. *IEEE Transactions on Computers*, 2023.

[6] Abhijitt Dhavlle, Rakibul Hassan, Manideep Mittapalli, and Sai Manoj Pudukotai Dinakarrao. Design of hardware trojans and its impact on cps systems: A comprehensive survey. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[7] Dina Mahmoud and Mirjana Stojilović. Timing violation induced faults in multi-tenant fpgas. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1745–1750. IEEE, 2019.

[8] Jiliang Zhang, Yaping Lin, and Gang Qu. Reconfigurable binding against fpga replay attacks. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(2):1–20, 2015.

[9] Han-Yee Kim, Rohyoung Myung, Boeui Hong, Heonchang Yu, Taeweon Suh, Lei Xu, and Weidong Shi. Safedb: Spark acceleration on fpga clouds with enclaved data processing and bitstream protection. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 107–114. IEEE, 2019.

[10] Mark Zhao and G Edward Suh. Fpga-based remote power side-channel attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 229–244. IEEE, 2018.

[11] Roel Maes, Dries Schellekens, and Ingrid Verbauwhede. A pay-per-use licensing scheme for hardware ip cores in recent sram-based fpgas. *IEEE Transactions on Information Forensics and Security*, 7(1):98–108, 2011.

[12] Tao Zhang, Jian Wang, Shize Guo, and Zhe Chen. A comprehensive fpga reverse engineering tool-chain: From bitstream to rtl code. *IEEE Access*, 7:38379–38389, 2019.

[13] Hoyoung Yu, Hansol Lee, Sangil Lee, Youngmin Kim, and Hyung-Min Lee. Recent advances in fpga reverse engineering. *Electronics*, 7(10):246, 2018.

[14] Ram Venkat Narayanan, Aparajithan Nathamuni Venkatesan, Kishore Pula, Sundarakumar Muthukumaran, and Ranga Vemuri. Reverse engineering word-level models from look-up table netlists. In *2023 24th International Symposium on Quality Electronic Design (ISQED)*, pages 1–8. IEEE, 2023.

[15] Holger Michel, Hipólito Guzmán-Miranda, Alexander Dörflinger, Harald Michalik, and Miguel Aguirre Echanove. Seu fault classification by fault injection for an fpga in the space instrument sophi. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 9–15. IEEE, 2017.

[16] Fabio Benevenuti, Fernanda Lima Kastensmidt, Ádria Barros de Oliveira, Nemitala Added, Vitor Ângelo Paulino de Aguiar, Nilberto H Medina, and Marcilei Aparecida Guazzelli. Robust convolutional neural networks in sram-based fpgas: a case study in image classification. *Journal of Integrated Circuits and Systems*, 16(2):1–12, 2021.

[17] Mohammad Shokrolah Shirazi, Brendan Morris, and Henry Selvaraj. Fast fpga-based fault injection tool for embedded processors. In *International Symposium on Quality Electronic Design (ISQED)*, pages 476–480. IEEE, 2013.

[18] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms. *arXiv preprint arXiv:1912.11523*, 2019.

[19] Mengmei Ye, Xianglong Feng, and Sheng Wei. Hisa: Hardware isolation-based secure architecture for cpu-fpga embedded systems. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[20] Shanquan Tian, Ilias Giechaskiel, Wenjie Xiong, and Jakub Szefer. Cloud fpga cartography using pcie contention. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 224–232. IEEE, 2021.

[21] Ilias Giechaskiel, Shanquan Tian, and Jakub Szefer. Cross-vm information leaks in fpga-accelerated cloud environments. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 91–101. IEEE, 2021.

[22] 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, 2018.

[23] Dennis RE Gnad, Cong Dang Khoa Nguyen, Syed Hashim Gillani, and Mehdi B Tahoori. Voltage-based covert channels using fpgas. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(6):1–25, 2021.

[24] Toshihiro Yamauchi, Yuta Ikegami, and Yuya Ban. Mitigating use-after-free attacks using memory-reuse-prohibited library. *IEICE TRANSACTIONS on Information and Systems*, 100(10):2295–2306, 2017.

[25] 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.

[26] Xilinx Vitis AI. https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/For-Edge-DPUCZDX8G/DPUCVDX8G.

[27] Xilinx Application Note XAPP1320: Isolation Methods. https://docs.xilinx.com/v/u/en-US/xapp1320-isolation-methods.