DmaAuth: A Lightweight Pointer Integrity-based Secure Architecture to Defeat DMA Attacks

Xingkai Wang, Wenbo Shen, Yujie Bu, Jinmeng Zhou, and Yajin Zhou, *Zhejiang University*

https://www.usenix.org/conference/usenixsecurity24/presentation/wang-xingkai

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

DMAAUTH: A Lightweight Pointer Integrity-based Secure Architecture to Defeat DMA Attacks

Xingkai Wang, Wenbo Shen[✉], Yujie Bu, Jinmeng Zhou, and Yajin Zhou

Zhejiang University

Abstract

IOMMU has been introduced to thwart DMA attacks. However, the performance degradation prevents it from being enabled on most systems. Even worse, recent studies show that IOMMU is still vulnerable to sub-page and deferred invalidation attacks, posing threats to systems with IOMMU enabled.

This paper aims to provide a lightweight and secure solution to defend against DMA attacks. Based on our measurement and characterizing of DMA behavior, we propose *DMAAUTH*, a lightweight pointer integrity-based hardware-software co-design architecture. DMAAUTH utilizes a novel technique named *Arithmetic-capable Pointer AuthentiCation* (APAC), which protects the DMA pointer integrity while supporting pointer arithmetic. It also places a dedicated hardware named Authenticator on the bus to authenticate all the DMA transactions. Combining APAC, per-mapping metadata, and the Authenticator, DMAAUTH achieves strict byte-grained spatial protection and temporal protection.

We implement DMAAUTH on a real FPGA hardware board. Specifically, we first realize a PCIe-customizable SoC on real FPGA, based on which we implement hardware version DMAAUTH and conduct a thorough evaluation. We also implement DMAAUTH on both ARM and RISC-V emulators to demonstrate its cross-architecture capability. Our evaluation shows that DMAAUTH is faster and safer than IOMMU while being transparent to devices, drivers, and IOMMU.

1 Introduction

Direct memory access (DMA) is a technique of computer systems that allows the peripheral device to access the main memory directly. DMA does not require CPU involvement during memory access and thus significantly reduces CPU occupancy. Therefore, DMA has been widely used on almost all computer systems, including large servers, personal computers, and mobile smartphones [12, 31].

However, the performance gain of DMA also comes with a price. DMA exposes the physical memory directly to the devices, bypassing all MMU-based protection (i.e., paging). All devices plugged into the host arbitrarily read/write the physical memory [9, 34]. To defeat DMA attacks, the Input/Output Memory Management Unit (IOMMU) has been introduced, which maps the physical memory to I/O virtual addresses (IOVAs) and gives the IOVAs to the peripheral devices [11]. Hence, the device cannot access the physical memory directly, and the access scope is limited to the mapped I/O pages. IOMMU is effective in defending against DMA attacks and has become the de facto approach for DMA protection.

However, recent research shows that IOMMU is not enabled on most systems, including Windows, Linux, and FreeBSD, due to performance concerns caused by IOVA translation and the slow IOTLB invalidation [12, 31]. Even when the IOMMU is enabled, the system protected by the IOMMU still suffers from both *spatial* and *temporal* attacks [3]. The page-grained-mapping nature of IOMMU allows a malicious device to access kernel objects that coexist on the same page with I/O buffers, leading to *spatial* sub-page attacks. The deferred IOTLB invalidation allows the device to access unmapped pages in the deferred time windows, leading to *temporal* deferred-invalidation attacks. Hence, how to defeat DMA attacks efficiently and thoroughly remains an open question.

To answer this question, this paper aims to provide a solution to address DMA security problems, which, in the meantime, is lightweight enough to be applied in an actual production environment. To achieve this goal, we first characterize the DMA behavior from devices of different types to see whether existing integrity-based or baggy-bounds-based methods are applicable. Our results show that 75.2% of requests are via DMA pointers plus an offset (termed the *pointer arithmetic*). At the same time, the number of coexisting mapped buffers is small (≤ 435), and the size ranges from 1B to about 520KB, while 68.7% of the buffer sizes are neither multiple-page-sized nor 2^n-sized. These results indicate that the existing methods are not feasible.

Based on measurement results, we propose *DMAAUTH*, a lightweight hardware-software co-design architecture that achieves byte-grained bound checking and strict temporal

✉ Corresponding author.

integrity on DMA pointers. DMAAUTH can thus defeat the DMA attacks that IOMMU suffers from, including the sub-page attacks and the deferred-invalidation attacks. In our design, a dedicated hardware named *Authenticator* is placed on the bus to authenticate all the DMA transactions, with the signature on the DMA pointers and corresponding metadata maintained by the OS kernel.

We face two challenges when designing and implementing DMAAUTH. First, the kernel loses control of DMA pointers once they are passed to the peripheral devices. In other words, devices can fake DMA pointers for memory access. Even with IOMMU, the malicious device can still forge an out-of-bound pointer to launch sub-page attacks or reuse outdated pointers to launch temporal attacks [3]. Second, protecting the pointer integrity while supporting the pointer arithmetic is difficult. Our characterization shows that benign devices frequently add offsets to given pointers and then dereference them for DMA. However, the existing pointer integrity-based techniques sign the whole pointer [28, 54] and are incompatible with peripheral pointer arithmetic.

To resolve the above challenges, we propose a new technique termed *Arithmetic-capable Pointer Authentication* (APAC), which leverages the pointer signing/authentication to grant the kernel complete control over the DMA pointers. APAC only signs the high bits and leaves the lower bits for arithmetic, overcoming the pointer arithmetic challenges. On top of APAC, DMAAUTH introduces per-mapping metadata and combines them to achieve spatial and temporal protection.

To evaluate the efficiency and effectiveness of the proposed techniques, we implement DMAAUTH on a real FPGA hardware board. Specifically, we first implement a fully functional SoC with customizable PCIe support on FPGA as our baseline. Based on our SoC, we implement the hardware Authenticator and integrate it into the PCIe bus to authenticate all PCIe DMA transactions. To enforce the protection, we protect the Linux DMA APIs to drive the Authenticator and realize DMAAUTH protection without changing device drivers. Additionally, DMAAUTH was also developed for RISC-V and ARM QEMU to demonstrate its cross-architecture capability.

We conduct both security and performance evaluations of DMAAUTH. The security evaluation shows that DMAAUTH can defeat all six types of DMA attacks, including the sub-page and the deferred-invalidation attacks. The performance evaluation on real hardware shows that DMAAUTH introduces a 1.0% throughput degradation and a 1.8% CPU runtime consumption overhead, outperforming the deferred IOMMU scheme by reducing throughput degradation and CPU overhead by 82.1% and 68.9%, respectively.

The contributions of this paper are summarized as follows.

- We characterize the DMA of different devices, providing a knowledge base of the DMA behavior.
- We propose a novel technique named APAC, which is the first pointer integrity protection scheme that supports pointer arithmetic.
- We propose a lightweight pointer integrity-based secure architecture named *DMAAUTH*, combining APAC and per-mapping metadata to thwart various DMA attacks.
- We develop a full-fledged PCIe-customizable SoC on real FPGA hardware, serving as a foundation for future architecture and system research.
- We implement DMAAUTH on real FPGA hardware based on our SoC. We also implement an emulator version of DMAAUTH for both RISC-V and ARM to show its cross-architectural capability. Our evaluation shows that DMAAUTH outperforms IOMMU in terms of both security and efficiency.

2 Background

2.1 DMA Workflow

DMA is a mechanism that transfers data without the full participation of the CPU to liberate it from the heavy burden of copying or moving data. Historically, DMAs were only configured by CPUs, but afterward, DMAs can also be issued by peripherals directly or via bus controllers. We use the Linux kernel as an example to illustrate the lifetime of a DMA transfer. We divide the whole process into three stages.

Pre-transfer is handled by the CPU and OS kernel. The `dma_map` API is called to map the kernel virtual address into the corresponding DMA pointer. `dma_alloc` API performs similar functionality but allocates a buffer in advance. Then, the kernel sends the DMA pointer to the peripheral and yields the CPU. If the IOMMU is enabled, the kernel is also responsible for maintaining the table for IOMMU address translation.

In-transfer is taken over by the peripheral and I/O bus. The peripheral launch DMAs and perform the actual data transfer via the bus. When finished, the peripheral informs the CPU by writing a message in memory or via interruption.

Post-transfer is handled by the CPU and OS kernel again. The DMA buffer is recycled using `dma_unmap` and `dma_free` API series. In compliance, the device should not reaccess the buffer. The IOMMU mapping for the buffer is also removed . Finally, `dma_free` also frees the DMA buffer after unmapping.

2.2 IOMMU

IOMMU was introduced into modern SoCs in the early 2000s to map only the specified pages to the device, preventing malicious devices from accessing unmapped memory [10]. However, IOMMU is still vulnerable due to the assumption that all the peripherals are parts of the trusted computing base (TCB) in OS implementations.

Memory management. IOMMUs use page tables to translate I/O virtual addresses (IOVAs) into physical addresses, similar to how MMUs translate virtual addresses. For each access from the peripheral using IOVA, the IOMMU hardware walks through its page table level by level and reaches the corresponding physical address. This scheme provides analogous protection capability as the MMUs; devices cannot access pages without mapping entry in the IOMMU page table but can still access contents residing on the same page, resulting in spatial *sub-page vulnerability*.

IOTLB cache. To accelerate the translation , an I/O translation look-aside buffer (IOTLB) is used to cache mappings from IOVAs to physical addresses. However, in the typical implementations, the consistency between the IOTLB and the IOMMU page table has to be maintained by the OS kernels via flushing the IOTLB after unmapping buffers [1,4,6,21]. Such flushes introduce significant overhead and are thus usually deferred by default to provide an acceptable performance. This allows peripherals to access unmapped pages in a time window, causing temporal *deferred invalidation vulnerability*.

PCIe ATS. To provide even better performance than the IOTLB integrated into IOMMU, SoC manufacturers provide address translation service (ATS) for trusted PCIe devices. Instead of translating the IOVA to the physical address on each peripheral memory access, the IOMMU informs the peripheral of the physical address after the translation. Then, the PCIe peripheral can choose to store the mapping from the IOVA to the physical address in its address translation cache (ATC), and it can launch DMA transactions using physical addresses directly [4, 6, 21]. But this mechanism exposes entire physical memory directly, causing *bypass vulnerability*. To make things worse, the Linux kernel trusts all the PCIe devices except those explicitly listed as external ports in the Advanced Control and Power Interface (ACPI) [15,52]. This allows all the PCIe devices plugged into the motherboard to use the ATS and access the main memory via bare physical addresses, positioning the entire system in danger.

2.3 DMA Attack

Computer system interconnects various components, including the powerful main processing units (such as CPU, GPU, and NPU), along with other accessory peripherals, which can be produced and marketed by untrustable manufacturers.

In the early years, without IOMMU protection, systems allowed all the peripherals to manipulate the main memory arbitrarily. During this period, DMA attacks were plain and powerful. Unlocking the OSes and executing arbitrary code is reachable via overwriting critical data in Windows [8], macOS [9, 13], FreeBSD [13], and Linux kernels [13, 14].

As discussed, IOMMU provides basic protection, preventing direct attacks. However, the high overhead introduced by IOMMU prevents the OSes from adopting it. Linux distributions often turn off the IOMMU protection by default, as did the different Windows editions [31]. The vulnerability of Linux further poses millions of Android devices in danger [12]. This fact reveals the demand for a more lightweight mechanism that can be applied in actual systems. Further research discovered that even when the IOMMU is enabled, the *sub-page vulnerability* and *deferred-invalidation vulnerability* still render the system vulnerable to DMA attacks [3, 31], urging researchers to design more secure protections.

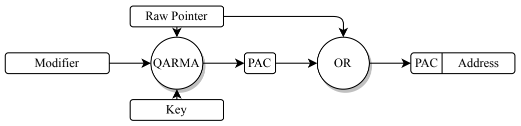


Figure 1: ARM Pointer Authentication. A hash signature is calculated by the QARMA encryption engine and embedded in the high bits of a pointer.

2.4 Pointer Authentication

Pointer authentication (PA) was first introduced into ARM v8.3 to ensure the integrity of pointers in AArch64 instruction set architecture (ISA). PA is an ISA extension that provides extra instructions for creating and authenticating authorized pointers. As shown in Figure 1, a signature is calculated based on the *pointer* value to authenticate, a 128-bit secret *key*, and a 64-bit *modifier*. Changing one of the three values results in a difference in the signature, preventing forgery of pointers. In this paper, we add a new pointer authentication instruction to meet specific pointer format requirements, but the methodology is the same as the ARM PA extension.

Signing a pointer. The ARM PA provides different keys for the different data categories: two for data pointers, another two for code pointers, and one for general data. For instance, to create a data pointer with a signature, `pacda <pointer>, <modifier>` is used. This instruction specifies the secret key using `d` for data pointers and `a` for the first key. The output signed pointer is stored in the original pointer's register.

Authenticating a pointer. To authenticate a pointer, the instruction `autda <pointer>, <modifier>` is used to recalculate the signature and strip it down from the pointer if the signature is valid. If the signature is invalid, it will be kept on the high bits of the pointer. This results in an illegal pointer and causes translation exceptions when dereferenced.

PA is the basis for many novel security-related works to ensure pointer integrity [22,28,47,55]. As the effectiveness and performance of PA have been tested in many scenarios, developers and researchers have been working on transporting the mechanism to other platforms [47,51,62].

3 Threat Model and Assumptions

In this paper, the attacker fully controls a DMA-capable accessory peripheral device. The attacker can read/write the main memory via DMA requests. When the IOMMU is absent, the attacker can forge fake DMA pointers and access all physical memory directly. When the IOMMU is enabled, the attacker exploits the sub-page vulnerability and deferred-invalidation vulnerabilities to access all mapped pages. The attacker aims to attack the OS kernel by launching DMA attacks, including memory dumping, kernel DoS, data pointer tampering, control flow hijack, information leaking, etc.

In this paper, we consider two system settings: the one with no IOMMU and the one with IOMMU enabled. We assume the main processing units(CPUs and computing accelerators, such as GPU), main memory, and the bus are trusted, and so are the enabled IOMMU. The OS kernel and device drivers running on the CPUs are trusted and use kernel DMA APIs correctly, as are the firmware running on the accelerators.

4 Characterizing DMA Behavior

We first characterize the DMA behavior of peripherals to guide the design of protection architectures.

4.1 System Setup

To monitor all the DMA buffers in the Linux kernel, we add logging logic to all the `dma_alloc` and `dma_map` series APIs of the Linux v6.1 kernel. Thus, all the DMA buffers' information can be recorded during the runtime, including the lower and upper bounds of the buffer, the device holding the buffer, and how many times the buffer has been mapped.

We intercept DMA transactions from different I/O buses, including PCI, PCIe, USB host controller interfaces (HCIs), etc. For each access, the start address, the length, and the access type will be recorded for further analysis. The Linux API modification only adds logging logic and does not impact the original DMA behavior. We use Buildroot to setup Linux with the following common peripherals: Intel E1000E NIC (`e1000e`), NVMe disk (`nvme`), Intel I/O Controller Hub 9 (`ich9`), QEMU Enhanced Host Controller Interface (`usb-ehci`), and other common USB peripherals like tablet, disks, keyboard and mouse are connected via the HCI interfaces. Once the emulation starts, our modification in the kernel and the emulator records all allocation, mapping, and accesses related to DMA, providing a better understanding of the DMA behavior.

We utilize two widely adopted network and storage benchmarking tools, `iperf3`[1] and `fio`[2], to generate authentic DMA traffic and furnish realistic workloads for characterization. On the one hand, `iperf3` conducts UDP send and receive operations with packet sizes ranging from 16B to 1460B, alongside TCP upload and download tests featuring window sizes ranging from 1KB to 1MiB. On the other hand, `fio` executes random and sequential read/write operations with block sizes spanning from 4KB to 4MiB.

[1]https://iperf.fr/
[2]https://github.com/axboe/fio

4.2 Access Characteristics

DMA consists of two parts: the mapping performed by the kernel and the data transfer conducted by peripherals. We analyze both parts to characterize DMA.

4.2.1 DMA Mapping Statistics

We analyzed the collected data to determine how many mapped buffers coexist in the memory. As shown in Table 1, the peripherals use a limited number of mapped buffers at the same time. We also read the driver source code of these devices to understand this behavior. The devices usually use I/O rings to hold descriptors for coexisting mapped buffers. Similar characterizing study for GPUs has been conducted [25] and reveals that the number of buffers used simultaneously by GPUs is also limited ($\leq$ 34 per kernel) and indexed by 14 bits. DAMN [33] shows the mapped *pages* are less than 12K.

Finding-1: Peripherals devices use a small number of coexisting I/O buffers (<500), despite that a large number of buffers had been mapped then unmapped (>5M).

We also analyzed the size of the mapped buffers, as shown in Figure 2. We find that the sizes of the mapped buffer vary significantly from 1 to 524288 bytes. The size of the buffers is very irregular. Only 30.2% of the buffers are multiple of page size, and only 28.1% are 2^n sized.

Finding-2: Peripherals' buffer sizes vary greatly, most of which are neither multipage-sized nor 2^n-sized.

The varied and irregular buffer sizes expose sub-page vulnerabilities and introduce obstacles to applying page-grained or 2^n-grained protection schemes like Baggy Bounds [2].

4.2.2 DMA Transfer Statistics

As shown in Table 1, peripheral devices often perform DMA transfer via the given DMA pointers plus an offset (termed pointer arithmetic). These accesses are mainly caused by the *I/O ring* scheme, in which the mapped buffers' descriptors are stored in a dedicated memory area. The device is responsible for reading the data from the area and launching DMA to the buffers specified by the descriptors.

Finding-3: Peripherals tend to apply pointer arithmetic on the given DMA pointers, accounting for 75.2% of all.

In our setup, these accesses are mainly fetching metadata from TX and RX buffers. In this scenario, the device continuously reads DMA-pointer-size data from a large buffer (usually one or more pages) for further DMA operations.

5 DMAAUTH Design

5.1 Goals and Challenges

To defeat DMA attacks and allow easy adoption by existing systems, DMAAUTH needs to meet the following design goals.

Table 1: DMA memory statistics.

Device Information		Pointer Arithmetic Statistics			Mappings Statistics	
Device	DMA Interface	With Offset	Total Access	Ratio	Coexisting Mappings	Total Mappings
NVMe SSD	PCIe (`nvme`)	4406751	5943096	74.1%	154	1487516
SCSI HDD	AHCI (`ich9-ahci`)	40	67	59.7%	13	15
Mouse and Tablet	EHCI (`ich9-ehci`)	40690	40956	99.4%	6	54
Keyboard	UHCI (`ich9-uhci`)	5066871	6629284	76.4%	5	32
USB Stick	EHCI (`usb-ehci`)	35086	35372	99.2%	5	33
E1000E NIC	PCIe (`e1000e`)	11230518	14985786	74.9%	271	3744537
Total	/	20779956	27634561	75.2%	435	5232187

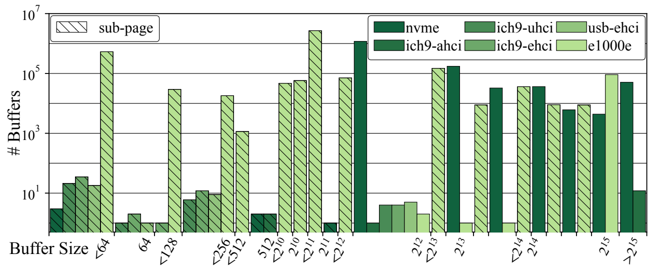


Figure 2: Sizes of the mapped buffers. The huge difference and irregularity of the sizes of the mapped buffers indicate that different buffers are used in highly different ways.

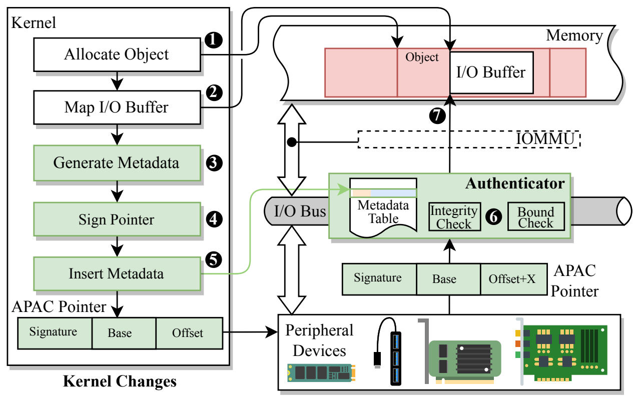


Figure 3: DMAAUTH workflow. Green-colored blocks are DMAAUTH components. DMAAUTH contains two parts: kernel changes and an Authenticator hardware on the I/O bus to intercept and check all DMA requests. DMAAUTH works with or without IOMMU.

Goal-1: Security. DMAAUTH must defeat both spatial and temporal DMA attacks. Particularly, DMAAUTH needs to defeat all sub-page attacks and deferred-invalidation attacks. Moreover, DMAAUTH must provide the same security guarantees without IOMMU or with ATS enabled.

Goal-2: Transparency. For easy adoption, DMAAUTH should require minimum changes on the OS kernels. Moreover, it should be fully transparent to the peripheral devices, device drivers, and the IOMMU DMA address translation process to provide excellent compatibility.

Goal-3: Lightweight. DMAAUTH should have minimum overhead to be adopted in the actual production environment.

To achieve the above goals, we need to resolve the following challenges to achieve DMAAUTH.

Challenge-1: No control on DMA pointers. The kernel loses control of the pointers once they are passed to the peripherals via MMIO. In other words, a malicious device can generate fake pointers arbitrarily and use them for memory access via DMA. Without IOMMU, the fake pointer can point to any physical address. Even with IOMMU, the malicious device can still forge out-of-bound pointers to access the kernel objects on the same page, leading to sub-page attacks [3].

Challenge-2: Pointer arithmetic compatibility. It is challenging to protect the integrity of the DMA addresses and support the pointer arithmetic. Our study in §4 shows that even benign devices frequently add an offset to the DMA pointers given by kernel drivers and use the new pointer with offset for DMA. Traditional pointer integrity-based technique is not compatible with pointer arithmetic. In other words, all these offset pointers will fail the pointer integrity check and are considered illegal. Therefore, we need to propose a new technique that can protect DMA pointer integrity and, at the same time, is compatible with pointer arithmetic.

5.2 Design Overview

To defend against DMA attacks, it is essential for the kernel to gain control of these DMA pointers, even after passing them to the device. To achieve this, inspired by ARM pointer authentication (PA), our key insight is that the kernel can sign all DMA pointers with secret keys that are inaccessible to the devices. When receiving DMA requests, the hardware Authenticator on the I/O bus checks the signature of DMA pointers. The devices don't have secret keys and thus cannot forge signed pointers. Therefore, the kernel has complete control of the DMA pointers and overcomes Challenge-1.

Unfortunately, ARM PA cannot be used directly due to the pointer arithmetic requirements. As shown in §4, devices often add or subtract an offset from a legal pointer and use it to access memory. ARM PA signs the whole pointer and does not allow any changes to the pointer. As a result, the pointers with pointer arithmetic fail the integrity check, disabling legal memory accesses. To solve this problem, we propose a partial pointer authentication technique named *Arithmetic-capable Pointer Authentication (APAC)*, which only signs the high bits and leaves the low bits for arithmetic, resolving Challenge-2.

Combining the above design and the technique, we propose DMAAUTH, which is a hardware-software co-design for defending against DMA attacks. More specifically, DMAAUTH uses the APAC technique to sign DMA pointers on the kernel side before passing them to peripheral devices. DMAAUTH also places the *Authenticator* hardware on I/O buses to intercept all DMA memory accesses, as shown in Figure 3. As APAC supports pointer arithmetic, DMA memory accesses by de-referencing the signed pointer directly or with offsets can pass the integrity checks, and the memory accesses are permitted. The forged DMA pointers or beyond-range offsets fail the integrity check, and the memory accesses are denied.

DMAAUTH workflow. DMAAUTH works with or without IOMMU. We use Figure 3 to illustrate how DMAAUTH works with and without IOMMU. Firstly, the kernel prepares the APAC pointer in the *pre-transfer* stage. ❶ When a device requires memory, the OS kernel allocates an object for I/O purposes, but only part of the object (the block mark I/O Buffer in Figure 3) should be allowed to be accessed by the device. But the rest of the object, along with the mapped page (colored red in Figure 3), is exposed to the peripheral, leading to sub-page attacks [3]. ❷ After memory allocation, the kernel explicitly maps the buffer to the page and gets the DMA pointer, which is a physical address without IOMMU, and an IOVA if IOMMU is enabled. ❸ According to the mapped buffer, the protected kernel generates the per-mapping metadata, which contains the bounds of the mapped buffer and a random identifier. ❹ Next, the protected kernel signs the DMA pointer using the whole metadata, to protect both the pointer and the metadata. The signed *APAC pointer* is then ready to be passed to the device. ❺ The metadata is stored in the corresponding entry, ready to be referred to by the DMAAUTH hardware. All metadata is kept in the Authenticator's dedicated memory, which can never be leaked even if the peripheral hijacks the data flow, providing extra resilience.

Then, the peripheral takes over the *in-transfer* stage and uses either the guarded pointer directly or with offsets as DMA addresses. ❻ When intercepting the DMA memory accesses, the Authenticator on the I/O bus first fetches the metadata, then uses the whole metadata entry to authenticate the pointer signature to determine whether the pointer is forged by a malicious peripheral or has been outdated. Moreover, the Authenticator further uses the bounds in the metadata to check whether the access is within legal ranges. ❼ Authenticator strips off the signature from the APAC pointer, forming a legal physical address or IOVA to proceed with the requested data access for the peripheral if the pointer passes the integrity and bound checks. This authentication process requires no participation of the CPU and the OS kernel. When the peripheral finishes the actual data transfer, the OS kernel continues the *post-transfer* stage and cleans the mappings.

Fulfill design goals. For security, DMAAUTH encodes the signature of the pointer into the high bits to form the *APAC pointer*, which is used to enforce the pointer integrity (§5.3). Moreover, DMAAUTH uses the bounds to defeat spatial attacks (e.g., sub-page attacks) and the identifier to defeat temporal attacks (e.g., deferred-invalidation attacks), as detailed in §5.4. Therefore, DMAAUTH outperforms IOMMU and defeats more DMA attacks. Besides, the key of DMAAUTH is stored in dedicated registers rather than memory and hence cannot be stolen by peripherals, thwarting pointer forgery attacks even when the adversary has access permission to the metadata. Once the metadata is stored in the Authenticator, it can never fetched and leaked to the peripheral, providing extra exploit tolerance. Therefore, DMAAUTH achieves Goal-1.

For transparency, when using table-gathered metadata, DMAAUTH requires less than 100 LoC changes in the Linux kernel and does not need to change the device driver at all. §6 shows that the Authenticator can be easily inserted between I/O buses and the memory to form a protected SoC design. Moreover, DMAAUTH is fully compatible with IOMMU. Therefore, DMAAUTH realizes Goal-2. DMAAUTH can be applied to different CPU architectures, such as ARM, RISC-V, and AMD64, as long as the QARMA is implemented.

For lightweight, §7.2 shows that DMAAUTH outperforms the deferred IOMMU scheme by ~80% in throughput and ~70% in CPU consumption, making it a better solution to be applied to the actual production environment.

5.3 Arithmetic-capable Pointer Authentication

As mentioned, we propose a novel pointer authentication technique named *Arithmetic-capable Pointer Authentication* (APAC) to protect the pointer integrity while allowing pointer arithmetic. The basic idea of APAC is to sign only the high bits and leave the lower bits for pointer arithmetic. In the following, we give details on the APAC-pointer format and the signing/authentication steps.

Pointer format. As shown in Figure 4, the APAC pointer consists of the following fields.

- A *signature* field holds the S-bit signature of the pointer.
- A *base* field to hold the 2^n-aligned lower bound of the mapped buffer.
- An *offset* field to hold the low bits of the addresses, acting as the offset within the buffer.

The *signature* length is S bits defined by SoC designers. The rest L bits are used to store the physical address, where $S+L=64$. The *offset* length is decided at the DMA mapping site. The protected DMA mapping API gets the size and DMA address of the mapped buffer, up-rounds the size to 2^n, and uses the n as the *offset* length stored in the metadata.

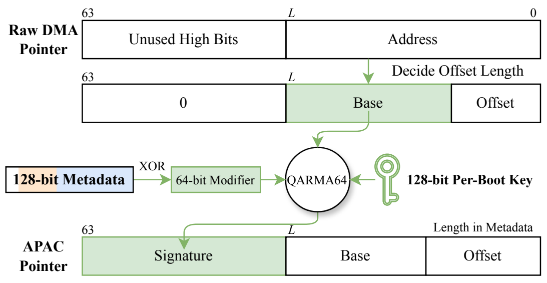


Figure 4: APAC pointer format. The signature of the pointer is calculated with the *base*, the *metadata* generated at mapping time, and the encryption *key*. The *offset* bits are not used for pointer authentication, thus supporting pointer arithmetic.

Pointer signing. The kernel performs the pointer signing during device memory allocation. As shown in Figure 4, After deciding the *offset* length, the DMAAUTH-protected kernel uses the *base* bits to sign and authenticate a pointer while skipping the *offset* bits. In this way, the devices can perform pointer arithmetic, which only changes the *offset* bits and doesn't impact the signature. Therefore, APAC pointers support pointer arithmetic while protecting the pointer integrity.

More specifically, the protected kernel takes the *base* field as the input of the QARMA encryption engine to generate the pointer signature. The other two inputs are the identifier in the per-mapping metadata and an encryption key. Note that the 128-bit metadata is transformed into a 64-bit modifier by XORing the high 64 bits and the low 64 bits to fit into the QARMA64 input. Next, DMAAUTH trunks the signature into *S* bits and embeds it into unused high bits of the DMA pointer. After these steps, the APAC pointer is ready to be transferred to the peripheral device for DMA purposes.

Pointer authentication. Pointer authentication is conducted by the DMA hardware when the peripherals launch the DMA memory access using the APAC pointers. DMAAUTH uses the signature in the pointer to locate and fetch the metadata from memory (detailed in §5.4). Once the metadata is fetched, the Authenticator gets the offset length and starts to authenticate the pointer to find out whether the pointer is forged by a peripheral, out-of-bound, or accessing unmapped memory.

If the peripheral device adds a legitimate offset to the APAC pointer, the *base* area of the pointer stays the same, and hence so is the signature re-calculated by the Authenticator. Once the pointer arithmetic results in overflow or underflow of the *offset* field, it will change the *base* of the APAC pointer. As a result, the whole pointer fails the authentication, and illegal memory access is denied. Using the APAC technique, the kernel has complete control over the DMA pointers.

Key management. DMAAUTH stores the key in both the CPU and the DMAAUTH hardware. In the CPU, DMAAUTH adds a key register to hold the key for the kernel to generate DMA pointers' signatures. In the Authenticator, the key is also stored to authenticate APAC pointers. During each boot time, a random value is generated and stored in the CPU key register using specialized instructions and in the key register of Authenticator hardware via MMIO write. Once initialized, these two registers holding the keys can never be accessed.

Physical address confidentiality. When using DMAAUTH only, the APAC pointer is based on a physical address. The confidentiality of the physical address can be achieved by adding a random offset to the physical address before signing the APAC pointer. The random offset is stored in the metadata and is used to recover the physical address when the APAC pointer is authenticated. When both DMAAUTH and IOMMU are enabled, confidentiality is guaranteed by the IOMMU.

5.4 Per-Mapping Metadata

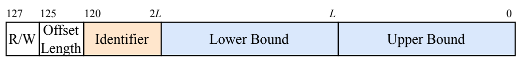
The kernel gains full control over the DMA pointers with the APAC technique. However, the unused bits in the pointer are not enough to encode a pointer's bounds information. While these bounds information are essential for spatial protection. To resolve this problem, our key observation is that the kernel uses memory DMA mapping/unmapping APIs to grant/revoke memory from devices. Therefore, we propose generating metadata for each memory mapping and storing the bounds in the metadata for spatial protection. Moreover, we also store a unique per-mapping identifier in the metadata to provide temporal protection. Note that we propose using per-mapping metadata rather than per-buffer metadata, as the latter cannot differentiate two mappings on the same buffer, leading to temporal attacks.

With the per-mapping metadata, when signing the DMA pointers, DMAAUTH fetches the metadata and uses it as the modifier for generating the signature. When the device accesses memory with APAC pointers, the Authenticator checks the signature to defeat any temporal attacks. Moreover, while performing the integrity check, DMAAUTH hardware further fetches the bounds from the metadata and performs a bound check to defeat any spatial attacks. Here resides the challenge to maintain and locate the corresponding metadata of an APAC pointer. In the following, we first discuss the metadata format and then give the metadata positioning scheme.

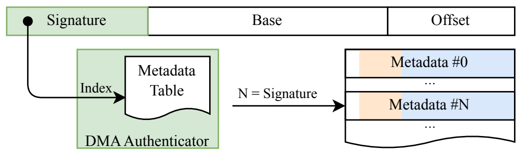
5.4.1 Metadata Format

Each metadata is 128-bit, consisting of 5 fields in Figure 5a.

- The *R/W* permission field. This field contains two bits and encodes three permission types: read-only, write-only, and bidirectional. The Authenticator checks this field to thwart unprivileged read or write on the mapped buffers for each DMA.
- The *offset length* field marks the length of *offset* field in the APAC pointer, which is 5-bit, supporting 4GiB buffer size. This field informs the DMAAUTH hardware



(a) The format of a per-mapping metadata entry.



(b) Metadata positioning. DMAAUTH uses the signature embedded in the APAC pointer to index the metadata in the metadata table.

Figure 5: Metadata format and the positioning scheme.

of the length of *base* field that should be taken into the authentication when using the pointer for DMA.

- The *identifier* is a $(121-2L)$-bit random number generated when the kernel sets up the mapping. This field also resolves signature hash collisions of APAC pointers.
- The L-bit *lower bound* of the mapped buffer.
- The L-bit *upper bound* of the mapped buffer.

The two bound fields provide byte-level bounds for at most L-bit physical address, allowing mapping any part of the physical memory region for the peripheral devices.

5.4.2 Metadata Positioning

As mentioned, the Authenticator intercepts the DMA and performs integrity and bound checking. Then, the APAC pointer is used to locate the metadata of the corresponding mapped buffer. Our key observation is that the number of the coexisting mapped DMA buffers is small (≤ 435 in Table 1.), which is suitable to be held in a dedicated metadata table.

As shown in Figure 5b, we integrate dedicated storage in the Authenticator to hold the metadata table and make the metadata table *write-only* to avoid metadata leakage. The APAC pointer holds S-bit signature and uniquely locates the metadata in the table, which holds 2^S metadata entries.

In the *pre-transfer* stage (§2.1), the DMAAUTH-protected kernel is responsible for filling the metadata of a newly mapped buffer into the metadata table and avoiding index collision. If the identifier causes index collision, another random identifier is generated. Finally, the metadata is written to the corresponding entry in the metadata table via MMIO.

Then, in the *in-transfer* (§2.1), the Authenticator uses the signature in the APAC pointer to index the metadata and uses it to authenticate the pointer and check the bounds.

Select signature length. SoC designers can easily customize the size of the metadata table to achieve a balance between circuit cost and supported coexisting buffers. For embedded devices that don't have to support massive coexisting buffers, the signature length can be set to 10. The corresponding metadata table holds 2^{10} entries, requiring only 16KiB extra SRAM storage. For desktop SoCs, the metadata table can be set to 2^{16} entries and requires 1MiB SRAM, which is small compared to the L3 cache (128MiB for Ryzen 7950X3D). For server SoCs, whose L3 cache is even larger (1.1GiB for EPYC 9684X), DMAAUTH can have up to 22-bit signature with 2^{22} entries, leaving 42 bits to support 4TiB DRAM.

Advantages. For each DMA memory access, the Authenticator requires only one extra memory access to authenticate an APAC pointer. In contrast, the common IOMMU has to access the main memory four or five times for each address translation. As a result, DMAAUTH has a significantly better performance compared with IOMMU-protected systems, as will be elaborated in §7.2. Moreover, DMAAUTH requires less than 100 LoC changes in the Linux kernel and requires no change on the device driver at all. Therefore, DMAAUTH is totally transparent to the driver developers.

5.5 Working Together with IOMMU

DMAAUTH is designed to be fully transparent to the IOVA translation process and thus is compatible with IOMMU.

IOVA translation. As shown in Figure 3, the Authenticator is placed closer to the peripherals than the IOMMU. The IOVA-based APAC pointers are generated the same way as physical-address-based APAC pointers. In the *pre-transfer* procedure, the metadata generation and pointer signing are conducted *after* the mapping is created by walking the IOMMU page table. In the *post-transfer* procedure, the signature is checked and stripped off by the Authenticator *before* being sent to the IOMMU for IOVA translation. The hardware and software co-design of the DMAAUTH is interposed *between* the IOMMU mapping-creating process and the IOMMU hardware translation process without changing the input of both two processes. In this way, we ensure the transparency to the IOMMU.

Providing ATS. The Authenticator hardware is designed to intercept the ATS request(§2.2) sent by the peripherals and return a corresponding APAC physical pointer. In the *pre-transfer* procedure, this requires the kernel to add the metadata of both the physical address and IOVA to the metadata table. To correctly translate an APAC IOVA to an APAC physical address, we use another table for signature translation.

The underlying limitation is that the mapped buffer using ATS has to be continuous in both the IOVA space and physical address space. This is already guaranteed in the `dma_map` APIs in order to use IOMMU. We just need to instruct the `dma_alloc` APIs to allocate physically contiguous memory using the `kmalloc` instead of `vmalloc`.

6 DMAAUTH Implementation

DMAAUTH contains the Authenticator implementation and the kernel changes, as shown in Figure 3. We implement two versions of the Authenticator: the FPGA-based hardware

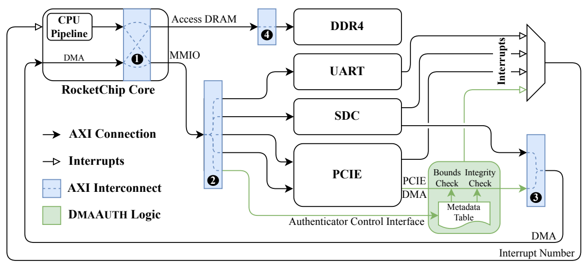


Figure 6: Hardware architecture of DMAAUTH. The green parts in the picture are the control and data path of the Authenticator. The blocks are connected via AXI protocol and are scheduled to transfer data by the AXI interconnects.

version and the QEMU-based emulator version. We implement DMAAUTH on Linux v6.1 kernel and implement the Authenticator to achieve fine-grained and transparent protection against malicious DMA-capable peripherals. The FPGA Authenticator is implemented with ~1500 lines of Verilog/Chisel. The emulator version adds ~200 lines of C code. The kernel changes are ~100 LoC with zero change on the device drivers.

We select the PCIe bus as our protected target for its high scalability and performance. We first construct a fully functional RISC-V SoC with PCIe 3.0 x8 based on the RocketChip core [7] as our baseline. We add pointer signing instruction `pacdma` to the core and Authenticator in Verilog to the bus, forming the DMAAUTH secure SoC prototype.

6.1 Authenticator Implementation

PCIe-customizable SoC implementation. To realize DMAAUTH on real FPGA, we first implement a fully-fledged RISC-V SoC with customizable PCIe support.The simplified schematic diagram is shown in Figure 6. The PCIe IP is officially supported by Xilinx and can be easily customized in Vivado GUI, making our design scalable for future research. However, the IP's lack of document support hinders it from being adopted by open-source SoC designs. We reverse-engineered the Xilinx PetaLinux[3] to configure the PCIe IP correctly, connect it to our SoC, and create the device tree.

The components in our design are connected using the AXI protocol, which is responsible for the *data transfer dispatching* and *cross-clock domain synchronization*. For data transfers, the AXI Interconnect ❶ receives access from the CPU and the DMA and determines whether it is access to the main memory or MMIO space, according to the address. If the address resides in the main memory region, the access is taken over by ❹ to read/write the main memory; otherwise, the access is determined as MMIO and guided to perform read/write access to the device registers by ❷. As the threat in our model, the PCIe device sends DMA requests to ❸, which can further forward the access to ❹ to access the DRAM.

[3]https://digilent.com/reference/software/petalinux

Synchronization between clock domains is also challenging since the design requires three different clock sources: CPU core clock, DDR4 clock, and PCIe clock. In particular, the PCIe clock requires a higher-quality differential signal, so we provide the signal with an external clock generator soldered on an FMC adapter. We use ❹ to synchronize between the CPU core and DDR4 DRAM. The synchronization between the CPU core and the PCIe is ensured by ❷ and ❸.

We use the `DefaultConfig` of the RocketChip generator, which has one `BigCore` with 16 KiB, 4-way set-associative instruction and data caches. The connection between the PCIe IP and the memory is limited to 100MHz with 64-bit data width to fully expose the throughput overhead introduced by the Authenticator. By intentionally making the host slower than the peripherals, we ensure that DMAAUTH acts as the system's bottleneck, facilitating a fair evaluation of its performance impact.

FPGA-based Authenticator implementation. As shown in Figure 6, the Verilog-implemented Authenticator is placed between the PCIe IP and next-level AXI interconnect to intercept and check all the DMA pointers from the PCIe bus. We decide to use 10-bit signature and 2^{10} metadata table entries because the single RocketChip core is more suitable for the peripherals in Table 1, which requires only ≤ 435 entries.

Firstly, the Authenticator has an AXI subordinate interface for the CPU to control it with MMIO. When the CPU accesses a specified range of addresses, it can write the key of the Authenticator and the metadata in the corresponding metadata entry in the metadata table, an FPGA block memory (BRAM) in our implementation. Notice that the CPU cannot directly read the metadata and key using MMIO to avoid metadata and key leakage. Especially, reading the occupied metadata entry returns `0x1`, and otherwise `0x0`, to allow the CPU and the kernel to resolve signature hash collision.

The Authenticator has another AXI subordinate interface to receive transactions from the PCIe IP. When the PCIe devices put Transaction Layer Packets (TLP) on the bus, the PCIe transforms the TLP into AXI access and then forwards it to the Authenticator. Notice that since the APAC pointers are transferred from the host to the peripherals, the peripherals use APAC pointers as target TLP addresses to perform DMA, and so are the AXI addresses sent by the PCIe IP.

Once the APAC pointer reaches the Authenticator, the authentication and bound checking described in step ❻ in Figure 3 is performed. The Authenticator has an AXI manager interface, which is responsible for forwarding the legitimate AXI transaction to the next-level AXI interconnect to perform DMA (step ❼ in Figure 3) and an interrupt signal to notify the CPU when the Authenticator detects a malicious DMA transaction. Once the interrupt is sent to the CPU, the corresponding trap handler in the protected kernel will further control the PCIe IP, forcing the peripheral to go offline.

We pipeline the authentication and data beats to reduce throughput overhead. Specifically, an AXI read burst consists

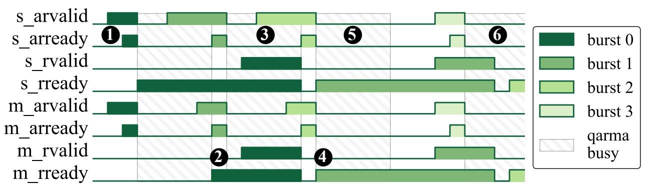


Figure 7: DMA read data transfer of the Authenticator. The diagram shows four AXI data bursts. The Authenticator receives DMAs launched by peripherals from the ports starting with `s` and forwards them to the next level via the `m` ports.

of two processes: address handshake and data beats, which can be overlapped and pipelined as shown in Figure 7.

We apply two AXI-related optimizations. First, when the peripheral starts address handshake by setting `s_arvalid`, the Authenticator sets `s_arready` to finish it if the QARMA engine is not busy, as indicated at ❶ in Figure 7. The `m_rready` is set to allow the data beats when the authentication is finished at ❷. Thus, clock cycles between ❶ and ❷ are used to prepare the required data. Second, the authentication of burst 1 (❸) is finished during the data beats of burst 0, and hence the `m_rready` can be set once the previous data beats are finished at ❹. So does the authentication of burst 2 and burst 3 (❺, ❻). These two optimizations pipeline bursts for rapid transfer.

IOMMU baseline implementation. To ensure fair comparison in evaluation, we introduce IOMMU to our SoC baseline. The RISC-V Foundation recently released its IOMMU specification [1]. But the standard hasn't been implemented in the RocketChip generator [7]. We extend the IOMMU on CVA6 [59] to our SoC and keep only a 1-level device table and a 3-level page table in order to ensure the baseline has the minimum translation latency required by the specification.

We config the IOMMU to have 256 IOTLB entries to ensure that the IOTLB pressure will not be the bottleneck for the system. The AXI-related optimizations are also applied to pipeline handshakes and data beats to achieve better performance. Additionally, the IOTLB flush will not block the CPU, saving ~2000 cycles CPU time [3, 5] for each flush.

After the above optimization, the IOMMU has only 5.6% overhead on transfer throughput and 5.8% CPU time overhead using deferred invalidation. This overhead is lower than commercially applied IOMMUs, which have ~10% overhead in throughput and ~10% in CPU time [33]. The IOMMU baseline without any logic other than address translation and IOTLB caching has the lower-bound overhead and is used as the baseline for the performance comparison with DMAAUTH.

QEMU-based Authenticator Implementation. We also implement the Authenticator on RISC-V and ARM64 QEMU to show its cross-architecture compatibility and IOMMU compatibility. QEMU uses `dma_memory_rw` to perform DMA data transfer. This API is responsible for translating the DMA pointer (IOVA when IOMMU is enabled) into a physical

```c
MemTxResult dma_memory_rw(AddressSpace *as, dma_addr_t addr, void
↪  *buf, dma_addr_t len, DMADirection dir, MemTxAttrs attrs) {
  dma_barrier(as, dir);
  dma_auth(addr, len, metadata);// Check signature and bounds.
  addr = lower_54_bits(addr); // Strip the signature.
  return dma_memory_rw_relaxed(as, addr, buf, len, dir, attrs);
}
```

Figure 8: APAC authentication in QEMU emulation. The signature is removed to make DMAAUTH transparent to the lower-level DMA logic.

address and read/write memory using it. *Before* the translation, authentication logic checks the access and strips off the signature on the high bits, as shown in Figure 8.

PAC Extension. To accelerate the APAC pointer signing process, we introduce pointer signing instruction extension `pacdma` to the RocketChip. `pacdma rd, rs1, rs2` takes the `rs1` as the pointer and `rs2` as the modifier to generate the APAC pointer and stores it to `rd`. The `rs1` is preprocessed to remove the *offset* part, and the `rs2` is preprocessed by XORing the higher 64 bits and lower 64 bits of the metadata. The QARMA engine is pipelined into 5 clock cycles to meet timing constraints and embedded as Rocket custom coprocessor.

6.2 Kernel Changes

The Linux kernel provides relatively uniform APIs for DMA operations, bringing convenience to our implementation. In general, the kernel uses the `dma_map` series APIs to grant access permission to the peripherals explicitly. As shown in Figure 9, the function checks whether the IOMMU is enabled and dispatches the DMA operation (Line 3-6) to call the direct helper function or creates an IOMMU mapping entry for IOVA translation. This dispatching corresponds to step ❷ in Figure 3. The driver uses the `dma_unmap` APIs to withdraw the access right from the peripheral. These functions also check whether the IOMMU is enabled and remove the translation entry from the IOMMU page table if IOMMU is enabled, as shown in Figure 10 (Line 5-8). Drivers utilize these APIs, so no modification of the driver source code is required.

Create Mapping. The `dma_map` takes an existing buffer as input and returns the corresponding DMA pointer as output. We let the API perform all the originally required operations. But before returning, the raw DMA pointer (physical address or IOVA) is signed with helper function `dma_auth_map` using the highlighted calls on Line 7 of Figure 9.

This metadata generation and pointer signing procedure is encapsulated in the helper function `dma_auth_map`. As depicted in Figure 3, the per-mapping metadata is generated (step ❸) in line 11 before calculating the APAC pointer (step ❹) in line 14. The helper function is also responsible for regenerating the identifier to resolve collision in the metadata table. Finally, in line 15, the metadata is inserted into the metadata table to be checked by Authenticator (step ❺).

```c
dma_addr_t dma_map_page_attrs(struct device *dev, struct page
↪  *page, size_t offset, size_t size, enum dma_data_direction
↪  dir, unsigned long attrs) {
  ...
  if (dma_map_direct(dev, ops)) // No IOMMU, direct map.
    addr = dma_direct_map_page(dev,page,offset,size,dir,attrs);
  else // IOMMU enabled, add mapping in IOMMU page table.
    addr = ops->map_page(dev, page, offset, size, dir, attrs);
  addr = dma_auth_map(dev, addr, size, attrs);
  return addr;
}
dma_addr_t dma_auth_map(struct device *dev, dma_addr_t addr,
↪  size_t size, unsigned long attrs) {
  struct metadata *metadata = dma_auth_metadata(addr,size,attrs);
  uint64_t base = dma_auth_mask(addr, metadata->offset_length);
  while (dma_auth_slot_taken(apac >> 54)) // If slot occupied.
    apac = dma_auth_apac(base, reshuffle_identifier(metadata));
  dma_auth_write_metadata(metadata, apac >> 54);
  return apac;
}
```

Figure 9: Generate an APAC pointer. The DMA pointer generated by the corresponding mapping function is an input to produce the signature with newly generated metadata.

```c
1  void dma_unmap_page_attrs(struct device *dev, dma_addr_t addr,
   ↪  size_t size, enum dma_data_direction dir, unsigned long
   ↪  attrs) {
2    const struct dma_map_ops *ops = get_dma_ops(dev);
3    addr = dma_auth_unmap(addr); // R/W=0, shuffle identifier.
4    ...
5    if (dma_map_direct(dev, ops))//IOMMU disabled,direct unmap.
6      dma_auth_unmap_page(dev, addr, size, dir, attrs);
7    else if (ops->unmap_page) // IOMMU enabled, clear mapping.
8      ops->unmap_page(dev, addr, size, dir, attrs);
9  }
```

Figure 10: Withdraw access right. After resetting the R/W bits and changing the identifier, any DMA corresponding to the metadata will be considered to be outdated and malicious.

Withdrawing Access Right. The access right towards the mapped buffer should be withdrawn immediately from the peripherals when unmapping the buffer, as shown in Line 3 of Figure 10. To achieve this goal, we disable all access using the outdated APAC pointer by resetting the R/W bits and rerandomizing the identifier. Without both the read and write permission, the peripheral has no access right to the unmapped memory region. This invalidation takes effect immediately, resolving the temporal vulnerability caused by the deferred invalidation. The direct unmap process or IOMMU-related unmap is called to clean up the mapping (Lines 5-8).

7 Evaluation

In this section, we evaluate the security and performance of DMAAUTH. The result shows that DMAAUTH defeats various DMA attacks while introducing low-performance overhead.

7.1 Security Evaluation

We evaluate the security capability of DMAAUTH using six DMA attacks of different types, including the sub-page and deferred-invalidation attack, as shown in Table 2. The evaluation results show that DMAAUTH can defeat all of them.

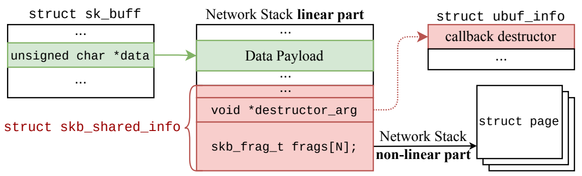


Figure 11: Memory layout of sk_buff structure. This data structure has been frequently exploited for sub-page attacks.

Table 2: Security evaluation.

Attack Information		Security Analysis		
Attacks	Type	Bare	IOMMU	Ours
❶ Full Memory Dump	Arbitrary Read	✗	✓	✓
❷ Denial of Service	Sub-page Write	✗	✗	✓
❸ Data Pointer Tampering	Sub-page Write	✗	✗	✓
❹ Control Flow Hijack	Sub-page Write	✗	✗	✓
❺ Information Leak	Sub-page Read	✗	✗	✓
❻ Access Unmapped	Temporal	✗	✗	✓

For the evaluation, we set up three settings based on Linux 6.1: one without IOMMU protection, one with IOMMU enabled, and one with DMAAUTH. We use an emulated malicious E1000E, which is adopted as the base of the Thunderclap platform to perform DMA attacks [31].

7.1.1 Defeating Spatial Attacks

We define a system to hold *spatial security* if the peripheral can only access the memory area that is mapped for it, i.e., realizing byte-level bounds to mitigate sub-page vulnerability.

Cases ❷ ❸ ❹ ❺ are the sub-page attacks that can be performed when IOMMU is enabled. All these attacks are implemented based on the structure sk_buff, a core implementation of the Linux network subsystem. As shown in Figure 11, the sk_buff struct carries data in its data field, which should be mapped to the NIC in streaming DMA. Different from previous works believing sk_buff is not an attack surface, further research [3] revealed that the data field of sk_buff has its memory organization with a linear part and a non-linear part. The descriptor of the non-linear part is stored immediately after the linear part in physical memory. These metadata structures are exposed to the NIC, making the DMA attack more powerful. In the following, we discuss these attacks in detail.

Case ❶: Full memory dump. The malicious device can easily dump the whole memory for the system without DMA protection. With IOMMU, the device can only access the mapped page, and this attack is defeated. DMAAUTH signs and authenticates all the DMA pointers and strictly restricts the pointers to their bounds, defeating the attack.

Case ❷: Denial of service. This attack is straightforward for a malicious NIC. It simply needs to blindly overwrite the entire page where the DMA buffer is located. Our experiments show that the host kernel always panics and crashes due to overwriting critical data pointers, even with the IOMMU.

Case ❸ Data pointer tampering and ❹ control flow hijack are similar. While the former one corrupts data pointers, while the latter corrupts the function pointer. More specifically, for the control flow attack, the attacker first forges a fake `ubuf_info` structure in a writable area, holding a callback pointer with a code pointer to the intended malicious gadget. Next, the attacker overwrites the pointer in `skb_shared_info` with the address of the fake structure. This way, the attacker can launch a control flow hijack to execute any kernel code. Sadly, these two attacks cannot be defeated by IOMMU.

Case ❺: Information leak. In this case, the attacker can exploit data pointers in `struct skb_shared_info` to leak important information, such as the kernel heap base and the base of the `vmemmap` segment. This information can bypass kernel ALSR. Again, IOMMU cannot prevent this leakage.

In our experiments, DMAAUTH can defeat the attacks in ❷ ❸ ❹ ❺. More specifically, the out-of-bounds reads and writes of DMA buffers are blocked by DMAAUTH's byte-level granularity (compared to the page-grained IOMMU). In this way, DMA is strictly restricted to the mapped buffer.

7.1.2 Defeating Temporal Attacks

As mentioned in §2.2, the OS kernels may sacrifice the IOTLB consistency to reach better performance in practice because of the unacceptable overhead introduced by IOTLB flushing [5, 26, 29, 50]. This brings deferred invalidation vulnerability to the system. As for the system without IOMMU protection, no temporal protection is provided, and the peripheral can access a specific memory area at any time.

Case ❻: Access unmapped memory areas. After a deferred invalidation, peripherals can still access the unmapped pages in a time window. Even if the driver checks all critical data after unmapping, there would still be a time window that malicious peripherals can tamper with some critical metadata after the checking. If the unmapped memory area is reallocated for other objects, the peripheral can also access the newly allocated object to perform further attacks. When using the strict mode, IOMMU can defeat such attacks but will introduce unacceptable performance overhead.

With DMAAUTH, the metadata update takes effect and invalidates the outdated APAC pointer immediately by re-randomizing the identifier and resetting R/W bits, and achieves temporal security with or without IOMMU.

7.1.3 Defeating Pointer Forgery and Substitution

The malicious peripheral may try to forge a valid pointer to bypass the pointer authentication conducted by the DMAAUTH. But without the 128-bit key stored in dedicated DMAAUTH and CPU registers, it can only try to break the authentication by iterating different signatures in brute force, which has a success rate of $\frac{1}{2^S}$ and requires 2^{S-1} attempts on average. Any failed attempts in the brute force break will trigger an exception to the CPU, notifying to disable the malicious device.

The substitution attack is prevented because the outdated APAC pointer is invalidated since unmapping. Because our metadata table is write-only, the attacker cannot obtain the metadata to perform substitution. The write-only metadata, random identifier, and R/W bits realize defense in depth.

7.2 Overhead Evaluation

We evaluate the performance of the DMAAUTH based on our FPGA prototype elaborated in §6.1. We use an FMC adapter to connect our SoC to the real peripherals: an E1000E NIC and an NVMe SSD. The workload used in performance evaluation is the same as §4. We also compare the hardware overhead of DMAAUTH with the IOMMU[4]. We noticed that the Rocketchip random generator is the bottleneck and causes large performance degradation, so we use `jiffies` as the random source to reduce the overhead.

7.2.1 NVMe SSD Performance

The NVMe SSDs are ubiquitous nowadays and are widely used in data centers and personal computers. We test the performance influence of DMAAUTH on the SAMSUNG 970 EVO Plus. With the tool `fio`, we test the sequential and random read/write performance with basic block sizing from 4KB to 4MiB by sending 1GiB of data in total with four threads.

Throughput evaluation. The throughput evaluation is based on the transfer speed the SSD can reach under different base block sizes. The results in Figures 12a and 12b reveal that the DMAAUTH introduces 1.4% overhead on average.

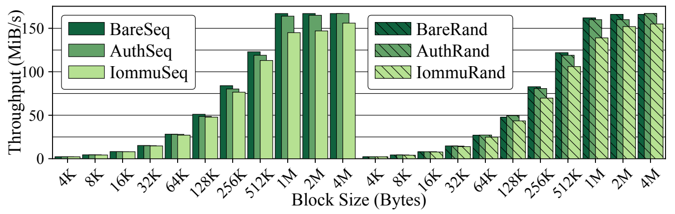
CPU runtime evaluation. DMAAUTH also introduces CPU runtime overhead for the metadata generation and APAC signing processes. We evaluate the cost by measuring the CPU time required to finish. The results in Figures 12c and 12d reveal that DMAAUTH introduces 1.0% extra CPU time.

7.2.2 Network Card Performance

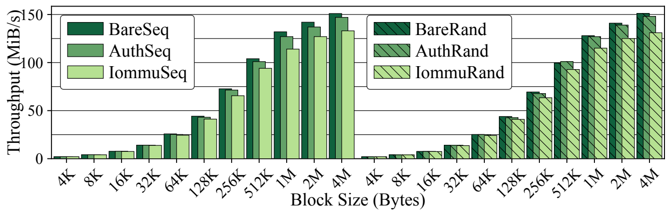
The Intel E1000E NIC has been proved to be an attack surface for DMA attacks [3]. We use a real E1000E card (Intel 82574) to test the performance influence of DMAAUTH on network cards. We use the network benchmark tool `iperf3` to send UDP packets from 16 to 1460 bytes to test UDP transfer performance; we also use this tool to perform TCP transfer tests in TCP window sizes ranging from 1KB to 128KB to reveal the performance degradation on the TCP stack.

Throughput evaluation. The TCP transfer throughput is shown in Figure 13a. The UDP transfer throughput is shown in Figure 13b. The DMAAUTH introduces 1.2% overhead in TCP throughput and $< 0.1\%$ overhead in UDP throughput.

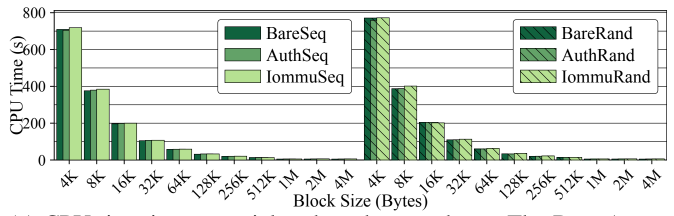
[4]https://github.com/zero-day-labs/riscv-iommu



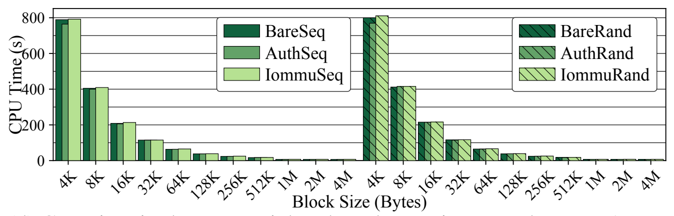
(a) Throughput in sequential and random read test. The DMAAUTH's average overhead is 1.3%, which is 82.7% less than IOMMU. The worst-case overhead of DMAAUTH is 8.6%.



(b) Throughput in sequential and random write test. The DMAAUTH incurs an average 1.4% overhead and is 79.3% less than IOMMU. The worst-case overhead brought by DMAAUTH is 3.8%.



(c) CPU time in sequential and random read test. The DMAAUTH operates with an average 1.1% overhead and is 81.9% smaller than IOMMU. The worst-case overhead introduced by DMAAUTH is 8.6%.



(d) CPU time in the sequential and random write test. The DMAAUTH carries an average 0.9% overhead, which is 81.2% less than IOMMU. DMAAUTH's worst-case overhead is 5.3%.

Figure 12: Performance evaluation on NVMe SSD. Combining the read and write test, the DMAAUTH introduces a 1.4% overhead in throughput and 1.0% overhead in CPU runtime on average, which are 81.0% and 81.6% less than IOMMU respectively.

Table 3: Hardware Resource Overhead.

Component	LUTs	FFs	BRAM
Bare SoC	69073	58605	58.5
DMAAUTH	4127 (5.9%)	4306 (7.3%)	4.0 (6.83%)
IOMMU	28432 (41.1%)	19477 (33.2%)	0

CPU runtime evaluation. We evaluate the CPU runtime head by measuring the CPU time required to finish transferring 1Gbits data. In the UDP transfer speed test, the CPU time equals the total transfer time, so it can be revealed in Figure 13b. The TCP runtime overhead is shown in Figure 13c. The DMAAUTH introduces 2.6% overhead in CPU time.

7.2.3 Comparison with IOMMU

As illustrated in §6.1, the IOMMU baseline ported to our SoC has 5.6% throughput overhead and 5.8% CPU runtime overhead, which is faster than the commercially applied SoCs' IOMMUs for its simplicity [11, 33]. Our evaluation shows that DMAAUTH only introduces an average 1.0% throughput degradation based on the NVMe SSD and E1000E NIC. The CPU runtime overhead is 1.8%. When the IOMMU operates in the deferred mode, DMAAUTH is 82.1% faster in throughput and 68.9% faster in CPU runtime.

We use Vivado[5] 2022.1 to synthesize the RTL and report circuit overhead and extra static power consumption as shown in Table 3. The DMAAUTH introduces 5.9% LUTs, 7.3% FFs, and 6.83% BRAMs overhead, while the IOMMU introduces 41.1% LUTs, and 33.2% FFs overhead when having 4 IOTLB, 4 DDTC and 4 PDTC entries in LUTs. DMAAUTH's BRAM overhead is due to the metadata table, which is used to store the per-mapping metadata.

[5]https://www.xilinx.com/products/design-tools/vivado

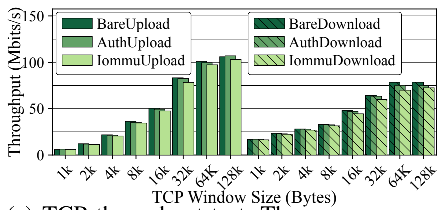
The SoC equipped with the DMAAUTH consumes 0.575W static power, which is 0.012W higher than the bare SoC. The extra power usage is about 2.1% of the bare SoC.

DMAAUTH requires on-chip storage for the metadata table, and the IOMMU requires on-chip storage to cache IOTLB and PTE. The SoC designer can adjust the size of the metadata table considering the number of DMA mappings according to the chip use case, as described in §5.4.2, to make the overhead acceptable compared with L3 cache or other on-chip storage. For the IOMMU, the size of the IOTLB and PTE cache should also be carefully chosen by SoC designers to balance the performance and overhead.

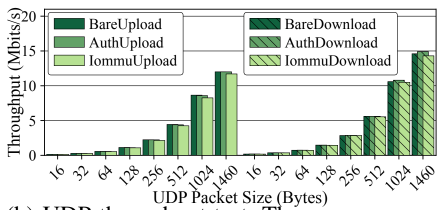
8 Related Work

DMA protection. DMA attacks have emerged for decades, but this threat is sometimes considered an out-of-scope problem or future work requiring research [16, 41, 45, 48, 57].

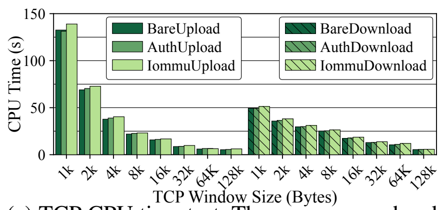
For systems with the basic protection of IOMMUs, there are works to thwart existing attacks using dedicated space for I/O operations [32], but this approach disables zero-copy and introduces significant performance overhead (20% CPU time and up to 25% throughput overhead). DAMN [33] kept zero-copy and reached better performance (10% throughput overhead). Kernel bypassing is enabled by DAMN to reach less CPU time than bare systems that disable such bypassing. However, the comparison when the bare system enables kernel bypassing is not conducted. DAMN's approach requires dedicated modification for each peripheral type and brings unacceptable manual workloads, making it not realistic to be applied in real-world OSes. These methods provide the same security guarantees as DMAAUTH but have larger overhead.



(a) TCP throughput test. The average overhead is 1.21% and is 76.5% smaller. The worst-case overhead of DMAAUTH is 4.8%.



(b) UDP throughput test. The average overhead $< 0.1\%$ while the IOMMU has a 3.1% overhead. The worst-case overhead is 1.8%.



(c) TCP CPU time test. The average overhead is 2.6%, which is 65.5% less than IOMMU. DMAAUTH's worst-case overhead is 9.0%.

Figure 13: Performance evaluation on E1000E NIC. Combining the TCP and UDP test, the DMAAUTH introduces 0.6% overhead in throughput and 2.6% overhead in CPU time on average, which are 85.3% and 65.5% less than IOMMU, respectively.

Fundamental modifications to IOMMUs are also proposed to realize find-grained protection. The novel hardware architecture is positioned into CHERI to enforce strong protection against DMA attacks [30]. But this requires radical modification towards existing hardware, and has not become available. Apple Inc. introduced byte-level protection into its IOMMU standard Device Address Resolution Map (DART) [42]. However, these solutions lack Linux support and face temporal vulnerabilities. Modifying hardware to make memory regions inaccessible for peripherals [35] is viable for embedded systems but not for time-tested complex OSes. AI-based methods are also introduced to defeat DMA attacks [19] but bring false positives and cannot provide deterministic protection. There have also been static and dynamic analysis tools detecting spatial and temporal vulnerabilities to help the developers improve memory hygiene [3].

Pointer integrity. Pointer integrity mechanisms are widely adopted in protecting embedded systems [43, 62], user space programs [28, 60], or the OS kernels [55]. It was first used to ensure control flow integrity by thwarting attackers from tampering with code pointers via software approaches [24, 43, 60, 62]. As new security features were introduced into modern computer systems, an increasing number of mechanisms were used to ensure pointer integrity, e.g., memory tagging [18] and pointer authentication [22, 28, 47]. Researchers are working on pointer integrity mechanisms to ensure better security [20].

However, the DMA attacks confront us with a distinct scenario: the CPU shall not participate in the actual data transfer procedure, making the methods mentioned above unavailable. These methods don't support pointer arithmetic and are thus incompatible with existing peripherals.

Memory protection. Corrupting memory was the oldest attack in the development of computer science and gave birth to various delicate exploit techniques and well-designed protection countermeasures [49, 58, 61].

To provide spatial protection, some mechanisms associate bound metadata with the objects and perform bound checking when conducting pointer arithmetic to restrict the pointer in the object [2, 27, 44, 56]. Some others are pointer-based and encode metadata into the pointers, using fat pointers [23, 39, 53] to store bound information or store it separately [17, 37], and check bounds when dereferencing the pointers.

Temporal vulnerabilities are mitigated by associating the allocation metadata with the allocated object [40, 46] or the output pointer [36, 38], but recording the metadata for each object fails to detect outdated pointer dereferences if the same memory area is reallocated for other objects.

Our mechanism is inspired by these memory-protection schemes, but it is not limited only to spatial or temporal. To provide spatial protection, it integrates the bound information in the metadata and uses redundant spatial information (*length* field) to provide extra protection and resilience. Also, each APAC pointer is associated with a mapped buffer and a write-only identifier via the signature encoded in the high bits to defeat temporal attacks and substitution attacks.

9 Conclusion

In this paper, we propose a lightweight pointer integrity-based security architecture named DMAAUTH, which achieves byte-grained protection on DMA memory and thus can defeat various DMA attacks, including the sub-page and the deferred-invalidation attacks. According to the findings from our characterization of DMA behavior, we design the novel Arithmetic-capable Pointer Authentication technique to ensure pointer integrity while allowing pointer arithmetic.

To verify the feasibility and performance of DMAAUTH, we realize the dedicated hardware Authenticator on the FPGA-based RISC-V SoC we implemented, which supports customizable PCIe. We also implement the Authenticator on the RISC-V and ARM QEMU to show DMAAUTH's cross-architecture capability. The implementation and evaluation prove that DMAAUTH outperforms IOMMU in both security and performance observably while staying transparent to IOMMU and device drivers.

Acknowledgments

The authors would like to thank our shepherd and reviewers for their insightful comments. Those comments helped to reshape this paper. This work is partially supported by the National Key R&D Program of China (No. 2022YFE0113200).

References

[1] Durbin Aaron, Baum Allen, Patel Anup, Pérez Daniel, Kruckemyer David, Favor Greg, Fawal Ahmad, Hunt Guerney, Hauser John, Scheid Josh, Evans Matt, Rodriguez Manuel, Kossifidis Nick, Donahue Paul, Walmsley Paul, Peresse Perrine, Tomsich Philipp, Ducousso Rieul, Nelson Scott, Zhao Siqi, V.L Sunil, Jeznach Tomasz, Papaefstathiou Vassilis, and Vedvyas Shanbhogue. *RISC-V I/O Virtualization Technology (IOMMU) Specification*. RISC-V Foundation, 2023.

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, volume 10, pages 51–65, Montreal, Canada, 2009. USENIX Association.

[3] 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*, pages 395–409, Online Event United Kingdom, April 2021. ACM.

[4] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification, 48882*. AMD, 2022.

[5] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. viommu: efficient iommu emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86. USENIX Association, 2011.

[6] ARM. *Arm System Memory Management Unit Architecture Specification*, 2023.

[7] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[8] Damien Aumaitre and Christophe Devine. Subverting windows 7 x64 kernel with dma attacks. *HITBSecConf Amsterdam*, 2010.

[9] Michael Becher, Maximillian Dornseif, and Christian Klein. Firewire: all your memory are belong to us. 01 2005.

[10] Muli Ben-Yehuda, Jon Mason, Jimi Xenidis, Orran Krieger, Leendert Van Doorn, Jun Nakajima, Asit Mallick, and Elsie Wahlig. Utilizing iommus for virtualization in linux and xen. In *OLS'06: The 2006 Ottawa Linux Symposium*, pages 71–86. Citeseer, 2006.

[11] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. The price of safety: Evaluating iommu performance. In *The Ottawa Linux Symposium*, pages 9–20, 2007.

[12] Gal Beniamini. Over the air: Exploiting broadcom's wi-fi stack, 2917. Accessed on May, 2023.

[13] Erik-Oliver Blass and William Robertson. Tresor-hunt: Attacking cpu-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, page 71–78, New York, NY, USA, 2012. Association for Computing Machinery.

[14] Rory Breuk and Albert Spruyt. Integrating dma attacks in exploitation frameworks. *Retrieved on January*, 14(2014):2011–2012, 2012.

[15] Jean-Philippe Brucker. Pci, iommu: Factor 'untrusted' check for ats enablement, 2020.

[16] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 379–393, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[18] R. Gollapudi, G. Yuksek, D. Demicco, M. Cole, G. N. Kothari, R. H. Kulkarni, X. Zhang, K. Ghose, A. Prakash, and Z. Umrigar. Control flow and pointer integrity enforcement in a secure tagged architecture. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1780–1795, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[19] Yutian Gui, Chaitanya Bhure, Marcus Hughes, and Fareena Saqib. A delay-based machine learning model for dma attack mitigation. *Cryptography*, 5(3), 2021.

[20] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan. ZerØ: Zero-overhead resilient operation under pointer integrity attacks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 999–1012, 2021.

[21] Intel. *Intel® Virtualization Technology for Directed I/O Architecture Specification*, 2022.

[22] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Tightly seal your sensitive pointers with PACTight. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3717–3734, Boston, MA, August 2022. USENIX Association.

[23] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[24] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, October 2014. USENIX Association.

[25] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. Securing gpu via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 27–41, New York, NY, USA, 2022. Association for Computing Machinery.

[26] Breno Henrique Leitao. Tuning 10gb network cards on linux. In *Proceedings of the 2009 Linux Symposium*, pages 169–185. Citeseer, 2009.

[27] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4211–4228, 2023.

[28] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. Pac it up: Towards pointer integrity using arm pointer authentication. In *USENIX Security Symposium*, pages 177–194, 2019.

[29] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 355–368, Istanbul Turkey, March 2015. ACM.

[30] A. Theodore Markettos, John Baldwin, Ruslan Bukin, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Position Paper:Defending Direct Memory Access with CHERI Capabilities. In *Hardware and Architectural Support for Security and Privacy*, pages 1–9, Virtual Greece, October 2020. ACM.

[31] 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*, San Diego, CA, 2019. Internet Society.

[32] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU Protection from DMA Attacks: When Copy is Faster than Zero Copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 249–262, Atlanta Georgia USA, March 2016. ACM.

[33] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. Damn: Overhead-free iommu protection for networking. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–315, 2018.

[34] Antonio Martin. Firewire memory dump of a windows xp computer: a forensic approach. *Black Hat DC*, pages 1–13, 2007.

[35] Alejandro Mera, Yi Hui Chen, Ruimin Sun, Engin Kirda, and Long Lu. D-box: Dma-enabled compartmentalization for embedded applications. *arXiv preprint arXiv:2201.05199*, 2022.

[36] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In *Proceedings of the 30th USENIX Security Symposium*, USENIX Security'21, August 2021.

[37] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[38] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 international symposium on Memory management*, pages 31–40, 2010.

[39] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, 2002.

[40] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[41] Wojciech Ozga, Rasha Faqeh, Do Le Quoc, Franz Gregor, Silvio Dragone, and Christof Fetzer. Chors: Hardening high-assurance security systems with trusted computing. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1626–1635, New York, NY, USA, 2022. Association for Computing Machinery.

[42] Sven Peter. Apple m1 dart iommu driver, 2021. Accessed on Month Day, Year.

[43] Jinli Rao, Tianyong Ao, Kui Dai, and Xuecheng Zou. Arce: Towards code pointer integrity on embedded processors using architecture-assisted run-time metadata management. *IEEE Computer Architecture Letters*, 18(2):115–118, 2019.

[44] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 2004, pages 159–169, 2004.

[45] Martin Schönstedt, Ferdinand Brasser, Patrick Jauernig, Emmanuel Stapf, and Ahmad-Reza Sadeghi. Safetee: Combining safety and security on arm-based microcontrollers. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 520–525, 2022.

[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. 2012.

[47] Gabriele Serra, Pietro Fara, Giorgiomaria Cicero, Francesco Restuccia, and Alessandro Biondi. Pac-pl: Enabling control-flow integrity with pointer authentication in fpga soc platforms. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 241–253. IEEE, 2022.

[48] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2541–2557, 2020.

[49] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[50] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. coiommu: a virtual iommu with cooperative dma buffer tracking for efficient memory management in direct i/o. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 479–492, 2020.

[51] Yu Wang, Jinting Wu, Tai Yue, Zhenyu Ning, and Fengwei Zhang. Rettag: Hardware-assisted return address integrity on risc-v. In *Proceedings of the 15th European Workshop on Systems Security*, EuroSec '22, page 50–56, New York, NY, USA, 2022. Association for Computing Machinery.

[52] Mika Westerberg. Pci / acpi: Identify untrusted pci devices, 2018.

[53] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 457–468. IEEE Press, 2014.

[54] Jinyan Xu, Haoran Lin, Ziqi Yuan, Wenbo Shen, Yajin Zhou, Rui Chang, Lei Wu, and Kui Ren. Regvault: hardware assisted selective data randomization for operating system kernels. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 715–720, 2022.

[55] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. In-kernel control-flow integrity on commodity oses using arm pointer authentication. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 89–106, 2022.

[56] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: An efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, page 145–156, New York, NY, USA, 2010. Association for Computing Machinery.

[57] Miao Yu, Virgil Gligor, and Limin Jia. An i/o separation model for formal verification of kernel implementations. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 572–589, 2021.

[58] Ziqi Yuan, Siyu Hong, Rui Chang, Yajin Zhou, Wenbo Shen, and Kui Ren. Vdom: Fast and unlimited virtual domains on multiple architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 905–919, 2023.

[59] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[60] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. Vtint: Protecting virtual function tables' integrity. In *NDSS*, 2015.

[61] Jinmeng Zhou, Jiayi Hu, Ziyue Pan, Jiaxun Zhu, Guoren Li, Wenbo Shen, Yulei Sui, and Zhiyun Qian. Beyond control: Exploring novel file system objects for data-only attacks on linux systems. *arXiv preprint arXiv:2401.17618*, 2024.

[62] Mohamed Tarek Ibn Ziad, Miguel A Arroyo, Evgeny Manzhosov, Vasileios P Kemerlis, and Simha Sethumadhavan. Epi: Efficient pointer integrity for securing embedded systems. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 163–175. IEEE, 2021.