

Address Translation Service for Confidential Computing

Supraja Sridhara
ETH Zurich

Paul Ferdinand Heinen
ETH Zurich

Melisande Zonta-Roudes
ETH Zurich

Anjo Vahldiek-Oberwagner
Intel

Rupin Vakharwala
Intel

Mona Vij
Intel

Shweta Shinde
ETH Zurich

Abstract—Confidential computing relies on trusted IO for secure and performant device access in untrusted cloud environments. However, enabling device-side Translation Lookaside Buffers (TLBs) via PCIe Address Translation Service (ATS) introduces security challenges. Thus, most cloud deployments disable ATS and suffer from performance loss. In this paper, we model the security requirements of ATS for trusted IO using a novel equivalence model between IOMMU translations and device-side ATS access. Leveraging this model, we analyze existing proposals that rely on multi-level permission tables, demonstrating both their security and opportunities for simplification. Building on this, we propose MOAT, a new defense that replaces memory-intensive checks with lightweight cryptographic authentication, ensuring secure and low-latency validation of pre-translated addresses. We implement both MOAT and improved permission-table-based defenses in a full-system cycle-accurate simulator. Our evaluation shows that MOAT improves performance by 12.93% over optimized permission tables while maintaining security, compatibility with unmodified ATS-enabled accelerator computation, and increases the device’s PCIe endpoint area by 0.58% only.

Index Terms—address translation service, confidential computing, trusted IO

1. Introduction

Confidential computing solutions such as Intel TDX, AMD SEV-SNP, and Arm CCA modify processor hardware and firmware to enable executing confidential virtual machines (CVMs) in the presence of an untrusted cloud service provider [1], [2], [3]. Trusted IO extensions such as Intel TDX-Connect, AMD SEV-TIO, and Arm RME-DA introduce further modifications to processors and devices for attaching devices to CVMs for direct access to CVM memory [4], [5], [6]. Trusted IO modifies the processor’s IOMMU to enforce run-time confidential computing guarantees such as limiting the device-side accesses to the CVMs they are attached to.

While address translations based on IOMMUs, similar to MMUs, provide strong security they are typically slow.

Translation Lookaside Buffers (TLBs), a well-known optimization, cache the translations and create a faster access path [7], [8]. However, when a large number of devices are connected to a single IOMMU, a single IOMMU TLB (IOTLB) shared by all the devices does not scale efficiently [8], [9], [10], [11]. The PCIe specification addresses this scalability challenge by moving TLBs from the host to the devices and adding an Address Translation Service (ATS) to the host IOMMU [12]. With ATS, devices can fetch and store translations in their device-side local TLBs. Then devices can use these pre-translated addresses to access host memory directly, bypassing slow IOMMU translations.

ATS’s performance gain comes with a significant trade-off: bypassing the IOMMU undermines the memory isolation guarantees of trusted IO. Thus, commercial deployments disable ATS by default for confidential computing [13], which results in a significant performance slowdown. To address this tradeoff, recent trusted IO specifications introduce mechanisms to enable ATS—the IOMMU checks if the ATS access adheres to the current page table mappings (e.g., by maintaining a multi-level permission table that maps physical addresses to the permissions that a device has when issuing a translation and then performing a permission table walk when checking ATS access). First, analyzing the security of these proposed ATS defenses is important. However, they are not yet available in production host CPUs and devices. Second, threat modeling of devices attached to CVMs is challenging. A CVM device is trusted only within the purview of the CVM that it is attached to, while all other co-resident CVMs and hypervisor do not trust such a device. Reasoning about the security of ATS defenses should account for this mutual distrust threat model.

Intuitively, any ATS defense that allows devices to bypass IOMMU translations must still check if the accesses adhere to the valid translations at that point in time, i.e., ensure equivalence between the fast (ATS bypass of IOMMU) and the slow path (IOMMU walk). In other words, any pre-translated access that is admitted by an ATS defense, would also have been admitted if the corresponding access was subject to the IOMMU translation process. We abstractly model the IOMMU translation and the behavior of a device attached to a CVM. Using the model, we can now analyze

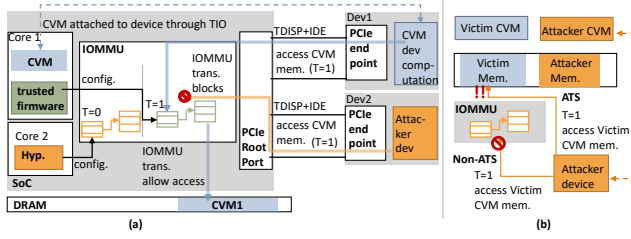


Figure 1. (a) Architecture with confidential computing and trusted IO. CVM executing on Core 1 is connected to Dev1 using trusted IO. Processor vendors change the IOMMU hardware to introduce protected IOMMU data structures that are configured by a trusted firmware. The IOMMU uses these trusted data structures to isolate device access (T=1) to CVM memory. Processor and device vendors change the PCIe root ports and endpoints to enable PCIe TDISP and IDE used to securely connect CVMs to trusted devices. (b) Device attached to attacker-controlled CVM uses ATS to bypass IOMMU translations and access victim CVM memory.

the security of existing proposals and new defenses for ATS.

As the first step, we analyze ATS defenses proposed by processor vendors that ensure security in the presence of ATS-capable devices with trusted IO. We successfully show that multi-level permission-table-based approaches by processor vendors are not only secure but can also be simplified to reduce memory footprint [6]. However, they store access control state in memory and perform a series of memory lookups to validate pre-translated accesses. This lookup chain degrades performance on critical ATS fast path even in non-trusted IO settings [14], [8], [9], [10]. A natural next step is to replace it with a classical alternative—embedding the access-control state in authentication codes (e.g., as done in pointer authentication) [15], [16], [17], [18]. The insight is to use lightweight cryptographic operations to check the authentication codes and validate memory access requests, instead of doing expensive permission table walks. Inspired by this paradigm, as a second step, we propose a new ATS defense named MOAT. Importantly, we show that our equivalence treatment to IOMMU securely refines MOAT design and optimizes performance.

For example, the dynamic nature of IOMMU translations, if not handled correctly, can compromise the temporal safety of an ATS defense. Specifically, when dynamic memory management adds or removes mappings, a pre-translated address that was valid when it was issued may become invalid when the device actually uses it. This problem is similar to invalidating stale TLB entries with a flush on mapping changes in the MMU. In case of ATS, as the TLB is on the device, the IOMMU cannot invalidate the mappings itself but has to delegate the invalidation to the device. Since we cannot trust the device to do the invalidation and revoke the authentication codes, we need a cheap mechanism on the IOMMU to detect stale addresses. Our insight is to change the cryptographic keys used to authenticate the pre-translated addresses. However, naively implementing this approach causes frequent shootdowns which are expensive. We observe that immediately updating keys is the main bottleneck. While there are several potential optimizations,

not all of them are secure (e.g., race conditions). We use our equivalence modeling to define an optimal yet secure variant, thus yielding better performance and confidence in the security claims.

We implement both ATS defenses (permission table and MOAT) on a cycle-accurate full-system simulator [19], [20]. Our optimization of the permission table reduces the table’s memory footprint by 6.25%. Further, for a device with 1024 TLB entries, our MOAT optimization that avoids TLB shootdowns reduces the amount of time spent waiting for host page table walks between 56.45% and 66.48% depending on the invalidation frequency. Overall, our measurements show that our intuition of replacing expensive walks with faster authentication checks yields an improvement of 12.93%. Next, we demonstrate the compatibility of MOAT with existing IOMMU and accelerator devices by implementing MOAT on an industry simulator. To evaluate the software compatibility, we execute unmodified accelerator-bound computation with Xilinx ATS. For hardware changes, we modify the host and accelerator with 619LoC and 455LoC, respectively [21], [22]. Lastly, MOAT induces a minimal area overhead of 0.58% for 128 TLB entries, which only impacts the device’s PCIe endpoint and not the acceleration units [23]. MOAT is open-source at <https://github.com/moat-cc> [24].

We make the following contributions in this paper:

- To check if an ATS defense is secure under trusted IO, we introduce the notion of equivalence between IOMMU translations and ATS. We use the equivalence to analyze proposed ATS defenses that perform permission table walks and securely optimize their memory footprint.
- We propose MOAT, an ATS defense that uses cryptographically generated and checked authentication codes.
- Our evaluation shows MOAT performs 12.93% better than optimized permission table walks. We demonstrate that MOAT is functional and compatible with existing software and does not induce large changes to hardware.

2. Background

We first provide background on IOMMUs and ATS and then explain confidential computing and trusted IO.

2.1. Security Implications of ATS

We explain the security implications of ATS accesses that bypass the IOMMU.

IOMMU and IOTLBs. In the cloud, CSP’s hypervisor configures IOMMU stage-2 translations from guest physical addresses (GPA) to host physical addresses (PA) enforcing access control on memory accesses from devices. In this setting, each VM OS can configure stage-1 translations from virtual addresses (VA) to GPA for its devices. With this, each memory access from the device is subject to slow memory translations making the IOMMU a bottleneck [14], [8], [9], [10]. Large IOTLBs in the IOMMU alleviate the performance bottleneck, but suffer from IOTLB contention.

Increase in devices and memory accesses results in a large number of IOTLB misses [8], [9], [10], [11], rendering the IOTLB ineffective.

PCIe Address Translation Service (ATS). ATS, an IOMMU bypass, is an optimal alternative to IOTLB. It allows devices to access host memory using pre-translated host physical addresses. To enable ATS, host IOMMUs provide support for devices to fetch translations from virtual addresses to host physical addresses [25], [26], [6] along with all other translation attributes (e.g., rwx permission, size, snoop). ATS-enabled devices add support to maintain and operate device-side TLBs which store the pre-translated addresses. Such a device can use a pre-translated address to issue read or write transactions of a device-specified length to host memory. The host IOMMU allows them through without translations.

Devices can abuse ATS to access arbitrary host memory. Consider an IOMMU connected to a malicious device with a compromised ATS implementation. This device can indicate to the host that the address in its request is pre-translated by setting the ATS flag in the memory requests. Now, the attacker can simply set the ATS flag and request access to any host PA and bypass the IOMMU checks. Prior works have shown that the attacker can set the PA in the request to any value to gain full access to host memory [27], [13].

ATS defenses. One way to prevent these attacks is for the cloud providers to disable ATS on all devices [13]. However, prior works have shown that disabling ATS significantly degrades the host-device PCIe bandwidth [28]. Intuitively, to prevent the above attack, we should ensure that devices can only use ATS to access memory regions that they could access if the IOMMU translations were enforced for each access. In this spirit, prior academic works and industry solutions to safely enable ATS change the host IOMMU and the device TLBs to ensure equivalence with IOMMU translations [29], [30], [25], [26], [6].

2.2. Confidential Computing and Trusted IO

In this section, we provide background on confidential computing and trusted IO, before explaining the threat model and its security implications.

Confidential VMs (CVMs). Confidential computing solutions like Intel TDX, AMD SEV-SNP, and Arm CCA enable executing sensitive computation in isolated confidential VMs (CVMs) [1], [2], [3]. They provide a cloud-native abstraction where users can lift-and-shift VMs to execute in a CVM isolated from an untrusted cloud service provider (CSP). Processor vendors use a combination of the modified hardware with trusted firmware to isolate the CVMs from CSP-controlled untrusted hypervisors and devices.

Native trusted IO. By default, devices cannot directly access CVM memory. Trusted IO solutions allow devices to directly access CVM memory by changing CPU and device hardware. On the CPU side, trusted IO in the form of TDX-Connect, SEV-TIO, and RME-DA allows CVMs to use confidential-computing-enabled devices [4], [5], [6]. On the device side, the PCIe specification includes a TEE

Device Interface Security Protocol (TDISP) and Integrity and Data Encryption (IDE) allowing devices to securely connect to CVMs [12]. Together, they ensure unforgeable identities for trusted devices and protect the data exchanged between the CVM and the device.

To isolate host memory accesses from devices, the PCIe specifications described above are necessary but are not sufficient. So, trusted IO solutions from processor vendors modify the processor's PCIe root port and IOMMU to isolate device accesses, as we explain next.

IOMMU changes for trusted IO. An untrusted CSP should not be allowed to configure the IOMMU for trusted devices attached to CVMs. To support this, trusted IO solutions modify the IOMMU hardware to introduce new protected per-device structures (e.g., translation tables, configuration tables) (Fig. 1.a) [6], [5], [4]. Only trusted firmware on the CPU can program these protected structures, which are inaccessible to the CSP hypervisor.

All device accesses to host memory are still subject to IOMMU translations. With PCIe TDISP, devices denote memory access to trusted CVM memory by setting a newly introduced trusted (T) bit in the PCIe TLP (Fig. 1.a) [12]. To decide whether to use protected IOMMU data structures to perform the translation, the IOMMU relies on the T bit propagated by the PCIe root port. If the T bit is set, the IOMMU recognizes that the memory access is to CVM memory, and uses the protected data structures to perform the translations. If the T bit is not set, the IOMMU uses the hypervisor-programmed IOMMU translation tables (Fig. 1.a) and does not allow these transactions to access CVM memory.

ATS with trusted IO. A device can use ATS to access CVM memory by setting the T bit when it fetches protected translations. As the T bit is set, the IOMMU uses the protected translation tables and returns the corresponding translation result to the device. Then, to access CVM memory using pre-translated ATS addresses, the devices set the T bit in the ATS request to indicate to the IOMMU that this is a PCIe transaction to CVM memory.

Threat model. We assume a standard threat model for confidential computing with trusted IO. We trust the confidential-computing-enabled hardware (i.e., processor) that the CVM runs on. We do not trust the CSP software (e.g., hypervisor) and any co-resident CVMs. Further, we do not trust the CSP's PCIe network (e.g., PCIe switches), and the PCIe endpoint devices (e.g., GPUs) that the CSP deploys. CVMs can use the trusted IO attestation process, to selectively trust devices. So, we trust the hardware of attested PCIe devices attached to the CVM within the scope of the CVM. Within the CVM scope, we do not trust all other devices in the data center. We assume that the CVM trusts all its software (e.g., OS, device drivers).

Attacker-controlled ATS devices. Attackers can use ATS devices to gain arbitrary access to host memory [27]. This threat is exacerbated under the confidential computing threat model with trusted IO, where a privileged adversary (e.g., the hypervisor) can launch an attacker-controlled CVM. The attacker can then execute the trusted IO attestation protocol within the CVM to attach an ATS device. Crucially, while

this device is attested and trusted by the attacker's CVM, it remains untrusted by all other co-resident CVMs. The attacker can leverage the CVM device to gain arbitrary access to other CVMs (Fig. 1.b).

3. Equivalence between IOMMU and ATS

For security, the host IOMMU should ensure that the devices cannot use ATS memory accesses to bypass IOMMU translations and compromise CVM isolation. To reason about potential defenses we abstractly model the IOMMU translation process and the behavior of an ATS device attached to a CVM. Then, in § 4 we use this model to analyze different ATS defenses and model MOAT in § 5.

IOMMU translations on the host. Consider an IOMMU with a translation state S (e.g., translation tables), and an untranslated access $(VA, attr_{in})$ where VA is the virtual address and $attr_{in}$ denotes the attributes (e.g., read/write, size, snoop) for the access. This access is allowed by the IOMMU, if the IOMMU translation which we denote using $IOMMU()$ returns OK . On successful translation, it also returns the physical address PA . To support ATS, the $IOMMU()$ optionally outputs $attr_{out}$ which denotes the attributes used for that translation (Fig. 3.a). Concretely, for all valid accesses,

$$IOMMU(VA, attr_{in}, S) = (OK, (PA, attr_{out})) \quad (1)$$

When the memory management of the hypervisor, CVM, or device changes the memory accessible to a device (e.g., allocate or deallocate), we capture this by updating the IOMMU state S to S' .

ATS on the device. After successful attestation of a device, a CVM trusts this device to adhere to the PCIe specification. The PCIe specification defines a benign ATS behavior that ensures equivalence with the IOMMU translation process, which we explain next. With ATS, devices can use pre-translated addresses to directly access host physical memory. Specifically, a benign device that complies with the PCIe specification invokes $IOMMU(VA, attr_{in}, S)$ in eq. (1) to fetch $(PA, attr_{out})$ from the IOMMU and stores it as the ATS_{state} for that translation in its TLB. From the specification, we know that a benign device uses ATS_{state} to send a pre-translated access with $(PA, attr_{in})$ where PA denotes the translated physical address. Eq. 2 represents a benign device functionality that sends a translated request with a physical address $(PA, attr_{in})$ if, and only if, the IOMMU had output the corresponding valid ATS_{state} with the PA . The CVM trusts the attested device (CVM_ATS) to exhibit this benign functionality when it accesses the CVM memory (PA_{cvm}). So, a CVM device will always satisfy the below property for all ATS accesses to CVM memory:¹

$$\begin{aligned} & \forall (PA \in PA_{cvm}, attr_{in}, ATS_{state}), PA \in ATS_{state} \\ & \wedge CVM_ATS(PA, attr_{in}, ATS_{state}) = OK \\ & \iff \exists (VA, S), \\ & IOMMU(VA, attr_{in}, S) = (OK, (PA, *)) \end{aligned} \quad (2)$$

1. $\iff$ denotes if, and only if; $=$ denotes output of the function.

where, CVM_ATS on PA_{cvm} is defined by the PCIe specification as the benign ATS behavior. For example, consider a physical address $PA_1 \in PA_{cvm}$ that a device only has read access on. Here, the ATS_{state} that a device retrieves for the corresponding virtual address VA_1 will store $(PA_1, attr_{out})$. For this translation, $attr_{out}$ will have read-only permission on that physical address. Now, if the device wants to read PA_1 , its $CVM_ATS()$ function returns OK satisfying eq. (2) and it sends a pre-translated read request to the host. However, the device will never send a write request to PA_1 , because CVM_ATS will not return OK as it would violate eq. (2).

Observe that eq. (2) only represents the benign ATS behavior when the CVM_ATS device accesses CVM memory (PA_{cvm}). This restriction holds because the CVM that attests the device trusts it to exhibit this benign behavior. However, the attestation process that a CVM executes only establishes trust within the scope of the attesting CVM. All other co-resident CVMs do not trust the device. So, a CVM_ATS device can exhibit rogue ATS behavior that violates eq. (2) when it accesses non-CVM memory. For example, the untrusted hypervisor that attested the device and attached it to a CVM it controls, can use the device to send arbitrary accesses to co-resident CVM memory. We capture this rogue ATS behavior on all non-CVM memory that violates the PCIe specification in eq. (3).

$$\begin{aligned} & \forall (PA \notin PA_{cvm}, attr_{in}, ATS_{state}), \\ & CVM_ATS(PA, attr_{in}, ATS_{state}) = OK \end{aligned} \quad (3)$$

In summary, CVM_ATS devices exhibit dual behavior: benign when accessing CVM memory (eq. (2)), and rogue when accessing all other memory (eq. (3)). To enable ATS securely with trusted IO, we need to reason about ensuring equivalence with the IOMMU translation process in the presence of CVM_ATS devices. We use our model of the IOMMU and the CVM_ATS device to reason about defenses and determine if they are secure in § 4 and § 5. Concretely, we analyze if a given solution Sol that relies on some Sol_{state} satisfies the below equation:

$$\begin{aligned} & \forall (PA, attr_{in}, ATS_{state}), \exists Sol_{state} \\ & Sol(CVM_ATS(PA, attr_{in}, ATS_{state}), Sol_{state}) = OK \\ & \iff \exists (VA, S), \\ & IOMMU(VA, attr_{in}, S) = (OK, (PA, *)) \end{aligned} \quad (4)$$

Observe that unlike eq. (2) which is only defined on CVM memory (PA_{cvm}), eq. (4) is defined on the full physical address space (PA). Therefore, eq. (4) captures the guarantee of Sol that ensures equivalence with IOMMU translations.

4. Existing ATS Support with Trusted IO

With SEV-SNP, AMD IOMMU blocks all ATS accesses to CVM memory from devices [25]. Arm CCA RME-DA and Intel TDX-Connect use permission table (PT) walks in the IOMMU to validate ATS access for devices.

4.1. Permission Tables (PT)

In this section, we analyze if *PT* ensures eq. (4) with CVM_ATS devices.

DPT with Arm CCA and HPT with Intel TDX-Connect. Arm’s Device Permission Table (DPT) is shared across devices and maps physical addresses to the devices and permissions that may access them [6]. In contrast, Intel’s Host Permission Table (HPT) is maintained per device and indexed using an unforgeable device ID [4]. Both tables are multi-level structures that allow them to scale to large physical address spaces and many devices. When a device is attached to a CVM, trusted firmware configures the corresponding DPT or HPT entries with the physical addresses and permissions for the device. On an ATS access, the IOMMU walks the appropriate table and allows the request only if the device is authorized to access the requested physical address.

Analysis. We observe that both Arm and Intel permission tables (*PT*) only check if the device has read or write access to the page and do not check all the IOMMU attributes ($attr_{out}$ in eq. (1)). We argue below that with confidential computing and trusted IO, this is sufficient to ensure equivalence.

We consider a CVM with a device CVM_ATS attached to it and $Sol = PT$ in eq. (4). We know that the CVM_ATS device always exhibits benign ATS behavior (eq. (2)) when accessing CVM memory (PA_{cvm}). Trusted firmware configures per-device PT_{state} as *PT* entries to ensure that the CVM device can only access the memory of the CVM it is attached to. So, *PT* only returns *OK* allowing an ATS access, if and only if the access is to a CVM address according to the *PT* entries captured by PT_{state} . Put together, this ensures that *PT* satisfies eq. (5) for all CVM addresses (PA_{cvm}).

$$\begin{aligned} & \forall (PA \in PA_{cvm}, attr_{in}, ATS_{state}), \exists PT_{state} \\ & PT(CVM_ATS(PA, attr_{in}, ATS_{state}), PT_{state}) = OK \\ & \iff \exists (VA, S), \\ & IOMMU(VA, attr_{in}, S) = (OK, (PA, *)) \end{aligned} \quad (5)$$

Therefore, for CVM_ATS devices it is sufficient to only check if the *PA* that the CVM_ATS is accessing is a CVM *PA* to ensure equivalence with IOMMU translations. This observation shows that the read/write permission checks in Arm DPT and Intel HPT are unnecessary.

Next, if the CVM_ATS device tries to access non-CVM memory, exhibiting rogue ATS behavior equivalent to eq. (3), *PT* will never return *OK* as there will be no valid mapping from non-CVM addresses to the device in the *PT*. Concretely, *PT* satisfies eq. (6) and blocks the access.

$$\begin{aligned} & \forall (PA \notin PA_{cvm}, attr_{in}, ATS_{state}), \nexists PT_{state} \\ & PT(CVM_ATS(PA, attr_{in}, ATS_{state}), PT_{state}) = OK \end{aligned} \quad (6)$$

From eq. (5) and eq. (6) we show that *PT* satisfies eq. (4).

Finally, when software adds/removes memory from a CVM it modifies the IOMMU page tables to grant/revoke the memory access to the devices (i.e., change *S* to *S'*) trusted firmware updates *PT* to reflect the new state *S'*. This prevents attacks from devices that might try to access revoked memory.

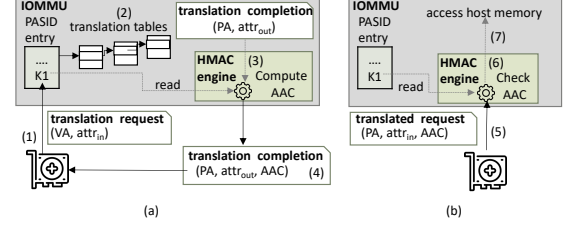


Figure 2. MOAT design. (a) Fetching a translation with AAC: 1. Device sends translation request to the IOMMU. 2. The IOMMU performs the translation and forms the translation completion. 3. MOAT HMAC engine intercepts the translation completion to compute the AAC. If the request contains a PASID, it uses the PASID key. If not, or if the Global bit is set, it uses the default key (§ 6.2). 4. Then, it sends it to the device to store. (b) Translated access with AAC: 5. Device sends the AAC with the translated access. 6. HMAC engine checks the AAC 7. and lets it through.

5. MOAT Overview

In the current permission table solutions, each ATS access triggers multiple memory lookups to walk the permission tables, where each memory lookup takes ≈ 100 cycles to walk page tables [31]. This can degrade ATS performance of Arm RME-DA and Intel TDX-Connect.

To replace the permission tables, in this section, we explore if we can use lightweight cryptographic checks on ATS accesses. Here, the originator of the access (i.e., the device) tags its request with a cryptographic token. Then, authentication checks on the host use lightweight cryptographic operations to validate the token and determine if an access is valid. Next, we present an intuition and abstractly reason about how MOAT performs such authentication checks on ATS requests to ensure equivalence with IOMMU translations. Then, § 6 presents MOAT design as a concrete realization of this reasoning, as summarized in Fig. 2.

Ensuring equivalence in MOAT. With MOAT, we need a $Sol = Moat$ which ensures that eq. (4) is satisfied, guaranteeing equivalence with a benign ATS device in the presence of a rogue CVM_ATS device. We observe that a benign ATS device uses the ATS_{state} returned in response to an ATS translation request to decide whether or not to originate a transaction. We encode this ATS_{state} in an unforgeable ATS Authentication Code (AAC), which serves as an authentication token. We return the AAC to the device as part of the response to a translation request. Then, when the device performs an ATS access it sends the AAC with the request, which we can easily check on the host without permission table walks.

Compute the AAC. We use a keyed cryptographic hash function (f_K) with a secret key *K*:

$$f_K(ATS_{state}) = (h, ATS_{state}) = AAC_h \quad (7)$$

where *h* is the cryptographic hash (we use AAC as a shorthand for AAC_h). So, with MOAT, when the device invokes IOMMU() to get translations, we send the AAC with it, i.e., (*PA*, $attr_{out}$, AAC) in eq. (1). The device stores

the AAC in its TLB and transmits it to the host on pre-translated accesses. On a pre-translated access, our check ($Sol = Moat$) on the AAC uses $Moat_{state} = K$. This check forces accesses from a CVM_ATS device to be equivalent to benign ATS behavior.

$$\begin{aligned} & \forall(PA, attr_{in}, AAC = (h, ATS_{state})), \exists K, \\ & Moat(CVM_ATS(PA, attr_{in}, AAC), K) = OK \\ & \iff \exists(VA, S), \\ & IOMMU(VA, attr_{in}, S) = (OK, (PA, *)) \end{aligned} \quad (8)$$

Checking the AAC. On every translated access $(PA, attr_{in}, AAC)$, the host IOMMU extracts ATS_{state} from the AAC, and verifies the hash using the secret key K (eq. (7)). MOAT only lets the translated access through if this check passes. So, MOAT ensures equivalence with the IOMMU translations by: (a) using the AAC to encode the ATS_{state} , and (b) checking it on pre-translated access.

The secret key K , ties the AACs to the device (§ 6.2), and ensures that a rogue device cannot forge AACs to trick MOAT check to satisfy eq. (8). With MOAT the device stores and transmits the AAC which contains the ATS_{state} . So, a rogue device can freely modify the ATS_{state} that it stores. However, even if the device modifies the AAC to ATS'_{state} , it cannot create a new hash h' for the new ATS'_{state} to satisfy eq. (8). So, the secret key ensures AAC unforgeability.

State stored in the AAC. We observe that we do not need to encode all of the ATS_{state} in the AAC to satisfy eq. (8). Here, we can assume a CVM_ATS device attached to a CVM. This device exhibits benign ATS behavior for accesses to CVM memory (PA_{cvm}) and rogue ATS behavior for all other memory regions (Fig. 1.b). So, if we constrain the device memory accesses to only PA_{cvm} using $Moat$ we can force it to always exhibit benign ATS behavior satisfying eq. (4) like the permission tables.

$$\begin{aligned} & \forall(PA, attr_{in}, AAC), \exists K, \\ & Moat(CVM_ATS(PA, attr_{in}, AAC), K) = OK \\ & \iff PA \in PA_{cvm} \end{aligned} \quad (9)$$

To satisfy eq. (9), we should create AACs that pass $Moat$ only for CVM physical addresses. We know that trusted firmware configures the IOMMU data structures to ensure that devices can only access their CVM memory regions. Hence, all valid ATS translation requests that pass the IOMMU() translation process have to be to CVM memory. In MOAT, we only generate an AAC using valid translation responses. This ensures that we only generate AACs for CVM memory that the device is attached to. Our $Moat$ in eq. (9) only needs information from ATS_{state} to determine if the device is accessing PA_{cvm} . We explain the exact information we encode in the AAC in § 6.1.

Handling invalidations. When software changes the translation tables in the IOMMU, i.e., transitions state S to S' , the IOMMU issues invalidation requests to the device to remove the affected device TLB entries. A benign ATS device will perform this invalidation such that eq. (2) is satisfied with the new state S' . In MOAT, we have to ensure that all old ATS_{state} encoded in AACs stored on the device, invalidated with the new IOMMU state S' , can no longer satisfy eq. (8).

For this, we can refresh the key K used as $Moat_{state}$ to K' on each invalidation. This key refresh ensures that any AAC created with K cannot satisfy eq. (8) that will now use K' . We observe that this naive approach effectively performs a TLB shutdown on the device, and the device has to re-populate its TLB on every invalidation request. To prevent this shutdown, we define an invalidation protocol, that allows the device to refresh all valid AACs in its TLB, before we change the key (§ 7).

6. AAC Enforced Isolation

To design MOAT, we analyze the IOMMU state in eq. (1) and select the fields in ATS_{state} for AAC computation to satisfy eq. (9). Then, we design our invalidation protocol around our observation that there is a time window between the state S and S' when mappings are changed.

Background: PCIe. PCIe packets, like network packets, consist of a header and a data section [12]. Optionally, these packets can also contain prefixes to support device-specific functionality (e.g., packet processing hints). Every PCIe packet header contains a device identifier. For packets requesting a read or write access to host memory, the header includes a length field specifying the number of bytes to transfer. Some devices support associating memory requests with Process Address Space ID (PASID), allowing CVMs (e.g., device drivers) to manage device memory access at a process granularity. In such cases, the device sends the PASID as a prefix to the PCIe packet, to associate memory requests with processes.

6.1. What Should an AAC Contain?

For ATS memory accesses $Moat$ uses the state in the AAC to ensure equivalence with IOMMU eq. (9). For this, the AAC should encode enough state to allow $Moat$ to validate that the ATS access is to CVM memory (PA_{cvm}). Next, we explain the fields of the result of the IOMMU translation (Fig. 3.a) that MOAT needs to encode in the AAC. **Physical address.** MOAT check needs to determine if the pre-translated address in an ATS memory request belongs to the CVM (i.e., in PA_{cvm}). So, we encode the PA from the IOMMU translation process in the AAC.

Encoding the size. The IOMMU returns both the base address and the size of the translation, as shown in Fig. 3.a. With ATS requests, devices can specify a length field indicating how many bytes to access from the base address. A malicious CVM device could exploit this to access memory beyond its assigned region by setting the length field to overflow into adjacent regions. While the PCIe specification prohibits devices from issuing memory accesses that cross the 4KB boundary, rogue devices can violate this and it is optional for receivers (e.g., on the host) to check [12]. Thus, under our threat model, we must check that access stays within the assigned region.

PCIe masks the lower bits of the translated PA to denote the size. We can use the masked translated address to compute the AAC to capture the size. However, to check



Figure 3. (a) Output of IOMMU() in eq. (1) with $(PA, attr_{out})$. AMA: ATS Memory Attributes. (b) MOAT data structure to create AACs. (c) AAC added as a prefix to PCIe ATS packets.

the AAC on a pre-translated access, we need to recompute this mask. So, we include the size as part of the AAC.

Global translations. The Global bit indicates that the translation is valid for all the PASIDs in the context. As we will explain in § 6.2, we optionally support creating the AACs using per-PASID keys. So, to maintain functionality when checking the AAC, we include this bit in our AAC.

Other attributes. Using the PA and size fields allows our check to isolate device access to CVM memory. Apart from these fields, the IOMMU translation process results in additional attributes shown in Fig. 3.a. We observe that these fields (e.g., NoSnoop, rwx permissions) are attributes that apply only to the region of memory that an ATS transaction is accessing. Once we have isolated the CVM_ATS device to only access CVM memory, we can trust the device to enforce these attributes on the device side as a benign ATS device would. Therefore, in MOAT we do not need to check or encode these attributes in the AAC.

6.2. AAC Key Management

Before explaining AAC key management, we provide some background on IOMMU data structures.

Background: storing device context in the IOMMU. To perform memory translations per-device, the IOMMU needs a mechanism to index into the device-specific page-tables. To support this, the IOMMU has data structures with device-specific information which contain pointers to the device translation tables. The hypervisor programs these data structures when new devices are connected to the VMs. Trusted IO introduces analogous protected IOMMU data structures that the trusted firmware programs.

Further, the IOMMU supports per-process memory translations. For this, the IOMMU device data structures point to PASID data structures that index into the translation tables programmed by the CVM OS. If there is a transaction without a PASID, the IOMMU uses a default PASID value. For performance, the IOMMUs deploy device table and PASID caches. We can trust trusted IO devices to perform process-level PASID isolation on the device side. Arm DPT does not support PASIDs whereas Intel HPT can be configured per-PASID.

Per-device AACs using per-device secret key. While we capture the attributes of the IOMMU translations in the AAC (§ 6.1), we do not capture device-specific information in the AAC. The IOMMU performs translations per-device by indexing into device-specific tables. We tie the AAC to

the unforgeable device ID using a per-device secret key (K in eq. (8)). To store the key, we extend the protected IOMMU device-specific data structures and use trusted firmware to program it when it attaches a device to the CVM. By reusing the protected IOMMU data structures to store MOAT secret keys, we ensure that the hypervisor cannot access them to forge AACs. Further, as the protected IOMMU data structures are accessed on trusted ($T=1$) transactions, using them ensures that we always use keys programmed by trusted firmware to check transactions that access CVM memory. Moreover, by reusing the IOMMU data structures MOAT benefits from the efficient caching effects that are already implemented in the IOMMU hardware for indexing into these data structures.

PASIDs. To improve how we handle invalidations (§ 7), trusted firmware can choose to tie AACs to their PASIDs instead of the entire device. Recall that we capture device-specific information in the secret key. So, just like using per-device keys to tie AACs to devices, the trusted firmware can use per-device, per-PASID keys to tie the AACs to a PASID within a device. The IOMMU performs PASID-specific translations for a given device, by first indexing into the device-specific table, and then indexing into PASID-specific tables. So, to support tying AACs to PASIDs, we extend the protected IOMMU PASID entries, and use the trusted firmware to manage the keys. Devices can originate requests without an associated PASID. In these cases, the IOMMU uses a default PASID value to index into the page tables for memory translations. To support creating AACs for requests without a PASID in MOAT, the trusted firmware programs the keys into the default PASID slot when the device is attached to a CVM.

6.3. Creating AACs

To create the AAC, we compute a keyed hash (HMAC) over the data structure in Fig. 3.b on each translation request.

TDX-Connect and RME-DA. On translation requests for CVM addresses, MOAT has to ensure that it uses the protected IOMMU data structures to create the AACs. TDX-Connect and RME-DA create a protected set of stage-1 and stage-2 translation tables that are programmed by CVM and firmware respectively. The IOMMU always uses these protected translations for requests to CVM memory. So, MOAT can use these translation results to create AACs.

SEV-TIO. AMD SEV trusted IO follows a slightly different approach. While CVM programs protected stage-1 translations, the untrusted hypervisor controls stage-2 translations in the IOMMU. SEV-SNP mitigates the risk of stage-2 translation mapping attacks from the hypervisor by using reverse map tables (RMP), which a trusted firmware programs. SEV-TIO extends these protections by subjecting the IOMMU translations to RMP checks. Consequently, the IOMMU subjects ATS translation requests to RMP checks before creating the translation completion response. So, in MOAT we can safely create the AACs after the RMP checks.

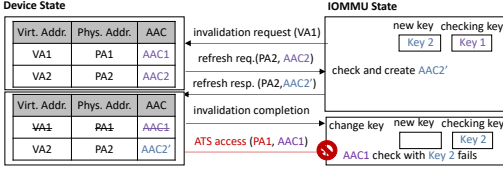


Figure 4. MOAT invalidation protocol allows devices to refresh the AACs.

6.4. Checks in the IOMMU

To check the AAC, the IOMMU first reconstructs the state in Fig. 3.b over which to compute an HMAC. First, it masks the physical address in the memory request with the size, and reconstructs the state. Then, it computes the HMAC over this reconstructed state by looking up the key based on the transaction’s PASID and the Global bit in the AAC (Fig. 3.c). Finally, it compares the recomputed HMAC with the one in the AAC to establish the AAC legitimacy. For the length check, the IOMMU reads the length of the access from the ATS request, and checks to see that the translation is valid for the whole range of accesses starting from the base physical address in the request. If all these checks pass, the IOMMU allows the access.

Reporting & handling AAC check failures. If the AAC checks fail at any point, the IOMMU reports a non-recoverable fault to the hypervisor. Typically, these non-recoverable faults require resetting or reinitializing the device to fix the faulty behavior. The hypervisor can report faults to the trusted firmware to take steps in a fault handler (e.g., stop CVM, reattach device). An untrusted hypervisor may ignore this fault and allow the CVMs to continue using the malicious device despite the ATS access check failures. This does not compromise MOAT’s security as the faulting device still cannot access any CVM memory. But the hypervisor can fake device faults to trick the trusted firmware. To allow the trusted firmware to validate the fault, the IOMMU reports the reason to protected memory.

7. MOAT: Ensuring Secure Invalidations

If we do not handle TLB invalidations securely, even with MOAT AAC checks, devices may gain unauthorized access to host memory.

Background: ATS invalidations. When the memory management (e.g., hypervisor, device drivers) updates the IOMMU page tables, it issues invalidation requests for the affected memory ranges to the device. With trusted IO, the trusted firmware (and CVM OS when it changes stage-1 translations) issues these invalidation requests. In response, a benign device invalidates the corresponding entries in its TLB. Once the invalidation is complete, the device notifies the host with an invalidation completion message.

Attacker abusing invalidations. Consider an attacker-controlled device that has legitimate access to a physical address PA1 attached to an attacker CVM. With MOAT, the device fetches a valid AAC for PA1, enabling it to access PA1

using ATS memory requests. Next, the hypervisor reassigns PA1 to a victim CVM and the trusted firmware sends an invalidation request to the attacker device to invalidate PA1. If the device were benign, it would invalidate the PA1 mapping and the associated AAC on receiving the invalidation request. However, under our threat model, the attacker device is not trusted by the victim CVM, and is considered malicious. So, the attacker device can exhibit malicious behavior and disregard the invalidation request, continuing to store the AAC for PA1. Now, when the victim CVM uses PA1 to write its own data there, the attacker device can access PA1 (read/write) using the still-valid AAC for PA1.

7.1. Potential Solutions

In MOAT we should prevent attacker devices from using stale AACs to gain access to revoked memory regions.

Option 1: TLB shutdown. We can refresh (change) the key in the IOMMU that is used to check the AAC. By changing the key, we ensure that any old AACs on the device, including those that were invalidated, can no longer pass the IOMMU checks. However, this is equivalent to a full TLB shutdown on the device. After each invalidation, device memory accesses will generate a TLB miss, triggering a translation request incurring performance overheads (§ 9.3).

Option 2: fixed-size invalidation list. Another approach is to store a per-device list of all mappings invalidated since the last key refresh [30], [29]. Then, for each ATS memory access check, the IOMMU checks the invalidation list before performing the AAC check. When the list is full, the IOMMU sends a TLB shutdown. So, if there is a large number of invalidation requests, this solution would only be as fast as a full TLB shutdown (§ 9.3).

7.2. MOAT Solution: Invalidation Protocol

To ensure that the devices cannot use invalidated AACs, we refresh the IOMMU key (as in Option 1). However, we should perform this key refresh in a way that does not degrade the device performance (e.g., effectively perform a TLB shutdown). We observe that the PCIe specification defines a window of time between invalidation requests and invalidation completions, during which the device can continue using old translations. The host IOMMU and software already account for this behavior and implement mechanisms to support it.

In MOAT, we can use this time window to refresh the AACs that are affected by the invalidation without disrupting the device operation. Specifically, the device executes an invalidation protocol to refetch the AACs that will be invalidated when the host refreshes the key in the IOMMU. While the device and host execute the invalidation protocol, the device can continue to use its pre-fetched translations without any disruptions to its operation. Then, once it has refreshed all the AACs, the device sends the invalidation completion. On receiving the invalidation completion, or if the invalidation request times out, the trusted firmware refreshes the IOMMU key invalidating the old AACs. So, the

invalidation protocol allows the device to continue operating without significant performance degradation.

Refreshing the key. For key refresh, MOAT creates two key slots in the IOMMU entry where it stores the keys. Trusted firmware programs the key into the new key slot before sending an invalidation request to the device. With this, the ATS checks can continue using the old checking key while the IOMMU and the device execute the invalidation protocol. In the meantime, the invalidation helper in the IOMMU uses the new key to refresh the AACs. Once completed, the IOMMU copies the new key into the checking key slot. This copy ensures that the IOMMU now uses the refreshed key, thus invalidating old AACs. This key update mechanism is not racy as the trusted firmware only writes to the new key slot, and the IOMMU only writes to the checking key slot. The IOMMU can always safely copy the new key into the checking key slot, as the trusted firmware only updates the new key slot once the invalidation protocol is complete.

Refetching AACs. If the trusted firmware creates AACs using a PASID-specific key, on an invalidation request, the device only has to refetch the AACs for that PASID. Before refreshing an AAC the IOMMU has to first validate if the AAC was valid. With MOAT, the device forms a list of (PA, AAC) pairs and sends the list in fixed-size chunks to the host. For each element in the list, the host IOMMU recomputes the AAC using the PA (see § 6.4) and checks that the AAC is valid and is not for addresses being invalidated. If the checks pass, then the IOMMU recreates the AAC using the trusted firmware-programmed new key (Fig. 4).

8. Security Analysis

We explain how MOAT mitigates attacks from both untrusted CSP and malicious co-resident tenants.

Fake or tampered AACs. A device can try to obtain an AAC for a memory region that it does not have access to by sending a translation request. MOAT will not create this AAC as the IOMMU translations for the request will fail. A device can also try to forge an AAC and send it with an ATS memory request to pass the IOMMU checks. Alternatively, a physical interposer can try to tamper AACs from IOMMU to the device. MOAT uses HMAC-SHA256 to compute unforgeable AACs [32], thwarting these attacks.

Attacks using invalidations. The device can try to abuse the invalidation protocol to gain unauthorized AACs which MOAT stops by first validating old AACs before issuing new ones. A hypervisor colluding with a malicious CVM can try to gain access to victim CVM memory by remapping it to an attacker CVM without issuing invalidation requests. Trusted IO stops this by routing all CVM memory mapping changes through trusted firmware, which enforces correct invalidations.

Attacker-controlled CVMs. An attacker can launch a CVM and try to access keys for devices attached to victim CVMs. Using these keys, the attacker can then use devices to create fake AACs and gain arbitrary access to host memory. This attack is infeasible, as hardware on the host prevents CVMs

from arbitrarily accessing the IOMMU data structures used to store the key [26], [25], [6].

Untrusted hypervisor. The untrusted hypervisor can try to read or reprogram the keys for the CVM devices in the IOMMU or fake AAC check faults to trigger handlers. MOAT prevents this by storing the keys and AAC check fault information in the protected IOMMU structures that are only accessible to trusted firmware.

Forge identity. An attacker can capture victim AACs using an interposer and try to forge device ID to access victim CVM memory. For trusted (T=1) transactions, trusted IO ensures unforgeable device IDs. For untrusted (T=0) transactions, MOAT checks will fail because all AACs to CVM memory are generated with keys from protected IOMMU data structures. MOAT only uses these data structures for trusted transactions.

Attacks on untrusted CSP. The hypervisor programs IOMMU translations to prevent attacks from rogue devices that try to access its memory. An attacker-controlled device can try to access hypervisor memory using ATS requests with T=0. With MOAT, such rogue ATS requests with T=0 are still subject to the AAC checks which are equivalent to IOMMU translations configured by the hypervisor.

9. Implementation and Evaluation

We implement MOAT changes to the Intel IOMMU in Qemu 8.1.0 with the Linux 5.4.0 kernel [21], [33] and modify Xilinx PCIe ATS simulator that uses SystemC 2.3.2 [22].

Intel IOMMU driver changes. The Intel IOMMU driver programs the IOMMU scalable mode PASID table entries with per-PASID information. We modify the IOMMU driver with 117LoC to allocate 512 extra bits per entry to generate and store MOAT keys for creating and checking AACs.

Host changes on Qemu. MOAT adds an HMAC engine to the IOMMU to generate and verify AACs. We extend Qemu with a cryptographic engine that uses OpenSSL's HMAC-SHA256 to compute AACs, when intercepting translation completions and translated accesses with 619LoC [34]. We implement MOAT AAC refresh protocol. Since invalidation requests only contain GPA ranges, the engine intercepts them, logs the GPAs, and forwards the request to the accelerator. Then, the engine translates the GPAs to PAs to compute corresponding AACs to invalidate. To support refresh, we add two new message types: AAC refresh request, and AAC refresh response. On receiving a request, the engine verifies each AAC, skips those marked for invalidation, computes updated AACs using the new key, and collates the refreshed values into the AAC refresh response.

Accelerator changes. To prototype the accelerator-side changes required by MOAT, we leverage Xilinx's ATS implementation, which models an accelerator using SystemC [22]. We extend the accelerator TLB to store AACs by adding 33 bytes to each TLB entry. Next, we update the handling of translation completions to store the AACs in the TLB. Further, we change translated memory requests to include the AACs from the TLB entries. We add sup-

TABLE 1. MOAT ATS REQUESTS IN FIG. 6 ACCELERATOR WORKLOADS

Benchmark	# Translation Requests	# Translated Accesses	Benchmark	# Translation Requests	# Translated Accesses
BlackScholes	23496	172032	Robot Localization	278719	622848
Denoise	3388143	3465728	Segmentation	2240265	2670844
Disparity	79928	1773276	Swaptions	232944	548352
EKF_SLAM	294957	737792			

port in the accelerator to perform AAC refreshes. Our total modifications amount to 455LoC changes.

Performance. We evaluate performance of MOAT by prototyping it on a cycle-accurate Parade full-system simulator with gem5 [20], [35], [19]. We introduce HMAC-SHA256 on the host to compute and check AACs on ATS requests. We modify the accelerator to simulate AAC store, fetch, and transmission. Next, we implement another setup inspired by Arm and Intel permission-table-based approaches, which we refer to as *PT*. Instead of the multi-level permission table, we prototype a more optimal single-level map from physical addresses to device IDs on the host and use it to check translated requests.

9.1. Functional Evaluation

We demonstrate the end-to-end feasibility of MOAT.

Qemu. Our modified Qemu faithfully simulates the IOMMU on the host and our PCIe ATS changes on the device. We measure the number of instructions during an accelerator job with an instruction tracer that uses the Qemu TCG Plugin API [36]. Qemu is not cycle accurate. So, we do not measure cycles on Qemu and only use it to validate MOAT functionality.

Experiment setup. We run our experiments on an Intel Xeon Gold 6246R host with 64 threads. Our host executes Ubuntu 20.04.6 LTS. We first execute Ubuntu with our Linux kernel changes on Qemu and connect it to our modified device. As a baseline, we use an unmodified Linux kernel with Ubuntu on Qemu and connect it to an unmodified device.

Single accelerator. MOAT changes add on average a total of 3805 instructions (0.2%). Of this, we see that MOAT adds 0.1% to the user-space, and 0.3% to kernel-space. We validate MOAT invalidation refresh protocol by sending an increasing number (10–1000) of TLB entries to refresh in the AAC refresh request. We report that the refresh protocol is feasible and the accelerator performs correct computation with the refreshes.

Multiple accelerators. Next, we validate MOAT functionality with multiple accelerators. We use 2, 4, 8, and 16 accelerators, and report that all accelerators compute correctly.

Proprietary industry simulator. We extend the IOMMU model on an industry simulator to store the keys and introduce an HMAC engine to generate and check the AACs. We modify an ATS-capable accelerator model to store the AACs in its TLB, and transmit them to the host and add support for the refresh protocol. We run the ATS and IOMMU test suites in the simulator to validate the functionality and compatibility of MOAT changes. We extend the ATS test

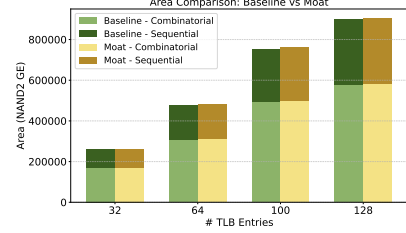


Figure 5. Hardware cost of MOAT and baseline. NAND2 GE: NAND2 gate equivalents reported by Stratus. Lower is better.

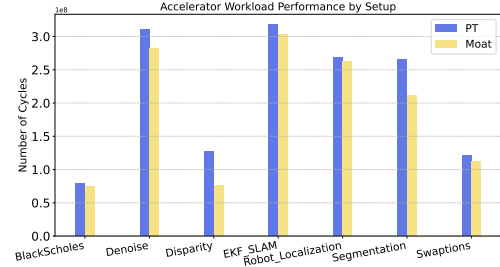


Figure 6. Accelerator workloads on Gem5 simulator. Lower is better.

suite to test MOAT functionality and model rogue accelerator behavior. Our tests show that MOAT changes preserve IOMMU functionality.

9.2. Hardware Costs

We estimate the cost of MOAT hardware changes to the accelerator using Cadence Stratus [23]. As a baseline, we port Xilinx ATS implementation with 128 TLB entries written in SystemC to Cadence Stratus. Next, we use our prototype from the functionality evaluation and port our SystemC modifications to Xilinx ATS implementation. Using Stratus, we synthesize the baseline and our changes to SystemVerilog and find that our changes add 1973LoC to the baseline of 80103LoC. From this synthesis, we observe that MOAT adds an area of 0.58% to the baseline with 128 TLB entries (Fig. 5). Further, we vary the number of TLB entries from 32 to 128 and find that MOAT area overhead is higher (2.08%) for an accelerator with only 32 TLB entries. As we increase the number of TLB entries, the percentage overhead of MOAT changes diminishes, because the total area required by the accelerator becomes the dominant factor (Fig. 5). We do not have an IOMMU hardware model, and so cannot estimate the cost of the hardware changes on the host.

9.3. Performance Evaluation

We measure MOAT performance using the Parade full-system simulator with gem5 [20].

MOAT performance. We compare MOAT performance to *PT* setup outlined in § 9. We use a benchmark setup with 32 TLB entries, on the Parade full-system simulator in

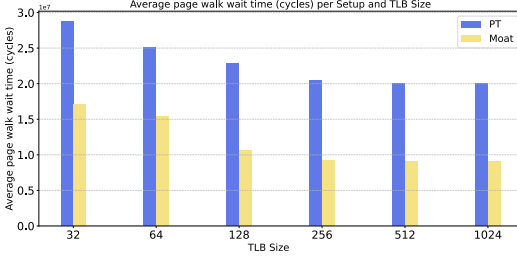


Figure 7. Host page walk wait time per device TLB size. Lower is better.

gem5 to perform our measurements [20], [19]. MOAT is on average 12.93% faster compared to *PT*. MOAT performance gains are the highest compared to *PT* when the device performs a large number of translated accesses compared to the translation requests (Tab. 1 and Fig. 6). We see the largest gains for MOAT in Disparity that performs 22× more translated accesses than translation requests. For Disparity, MOAT is 40.43% faster compared to *PT*. Next, we compare the number of cycles that the accelerator waits for host page table walks. We see that MOAT waits 51.93% less compared to *PT* on host page walks.

To assess TLB size impact, we run benchmarks with a fully associative TLB, varying entries from 32 to 1024. MOAT speedup compared to *PT* increases from 12.93% for 32 TLB entries to 14.95% for 1024 TLB entries. We see a stark difference between the amount of time MOAT spends waiting for host page walks compared to *PT* (Fig. 7). MOAT waits on average 51.93% less compared to *PT* with 32 entries. With 1024 entries, the wait time reduces further, with MOAT waiting 58.85% less compared to *PT*.

Invalidation. By default, the Linux kernel sets the invalidation timeout to 1 second. By conservatively considering a 1ms invalidation timeout, our calculations accounting for PCIe bandwidth and AAC check and computation latency show that the device can refresh up to $\approx 48,000$ TLB entries without any overheads. Next, we measure the performance of the two potential solutions in § 7.1. First, like prior works, we assume 200 invalidation requests per second and use a small TLB with 32 entries [30], [29]. With this, MOAT’s page walk wait time reduces by 56.45% compared to our first option (§ 7.1) that invalidates the device TLB on every invalidation request. Compared to the second option, MOAT’s page walk wait time reduces by 51.97%. We see that increasing the TLB size to 1024 entries further reduces the wait time of MOAT by 63.34% and 53.92% compared to the two options respectively. Next, we increase the number of invalidation requests to 1000 invalidation requests per second. With this, we find that the MOAT wait time reduces by 66.48% and 61.21% compared to the first and second options respectively.

Memory requirement. MOAT needs an additional 512 bits of memory in the host IOMMU per PASID and 263 bits per TLB entry on the device. In contrast, the memory for Arm DPT depends on the size of physical memory on the host and only covers confidential memory. For host memory of 1TB

and 4KB pages, Arm DPT requires ≈ 4 GB. Equivalently, MOAT would need 512 bits per PASID and covers the full memory space. With 1K PASIDs per device, MOAT requires only 64KB per device, and supports approximately 64K devices within the same 4GB budget. Our optimization of Arm DPT can save up to 1 byte per entry, resulting in 6.25% memory footprint reduction.

10. Related Work

ATS attacks and defenses without trusted IO. Thunderclap exploits ATS to bypass the IOMMU, enabling malicious devices to access memory illegally [27]. In response, the Linux kernel patched ATS support to disable it entirely or restrict it to a set of trusted devices [37], [38]. Border Control uses a single-level reverse map, which records translated addresses and their corresponding page permissions [29]. CryptoMMU improves on Border Control performance by adding MAC-based checks to the IOMMU for ATS requests [30]. However, these solutions do not reason about the confidential computing setting with trusted IO. Retrofitting these solutions for trusted IO is non-trivial and would require the comprehensive modeling and analysis that we present. MOAT is the first work to reason about ensuring IOMMU equivalence on ATS accesses, model the ATS device, and concretely demonstrate that we can use cryptographic checks for security in the trusted IO setting. Formal verification of data structures like page tables and permission tables with dynamic changes is its own undertaking [39], [40], [41], [42].

ATS with trusted IO. Prior works that attach devices to CVMs do not reason about ATS and always require IOMMU translations to isolate device accesses to host memory [43], [44], [45], [46]. Because MOAT ensures equivalence with IOMMU translations, MOAT check is compatible with these designs and can be used with them to securely enable ATS. Cure extends the bus protocol and tags memory transactions from integrated devices attached to trusted execution environments [47]. However, while Cure addresses integrated devices, MOAT design applies to PCIe devices.

CXL. Compute Express Link (CXL) and its trusted IO support build on PCIe to support high-speed memory-coherent host–device communication [48], and CXL.cache depends on ATS for coherence. These advances make MOAT increasingly important for future accelerator usability.

11. Conclusion

We present a principled approach to ATS defenses for trusted IO by modeling the equivalence between IOMMU translations and pre-translated device accesses. Building on this foundation, we propose MOAT, a lightweight cryptographic authentication-based ATS defense. Our evaluation shows that MOAT outperforms permission table defenses while remaining compatible with existing software. As confidential computing rolls out with hardware changes on host and devices in the coming years, MOAT provides a well-aligned and timely path to enable ATS for trusted IO.

Acknowledgments

We thank the anonymous reviewers for their feedback. We thank Mark Kuhne, Andrin Bertschi, and Clément Thorens for their helpful comments and assistance with proofreading and improving the paper.

References

- [1] Intel, “Intel Trust Domain Extensions (Intel TDX),” <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>, 2025.
- [2] Arm, “Arm Confidential Compute Architecture (ARM-CCA),” <https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture>, 2025.
- [3] AMD, “AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More,” <https://docs.amd.com/v/u/en-US/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more>, 2020.
- [4] Intel, “Intel TDX Connect Architecture Specification,” <https://www.intel.com/content/www/us/en/content-details/773614/intel-tdx-connect-architecture-specification.html>, 2025.
- [5] AMD, “AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization,” <https://www.amd.com/content/dam/amd/en/documents/developer/sev-tio-whitepaper.pdf>, 2023.
- [6] Arm, “Arm System Memory Management Unit Architecture Specification SMMU architecture version 3,” <https://developer.arm.com/documentation/ih0070/latest/>, 2025.
- [7] T. Merrifield and H. R. Taheri, “Performance Implications of Extended Page Tables on Virtualized x86 Processors,” *SIGOPS Oper. Syst. Rev.*, 2017.
- [8] N. Amit, M. Ben-Yehuda, and B.-A. Yassour, “IOMMU: Strategies for Mitigating the IOTLB Bottleneck,” in *International Symposium on Computer Architecture (ISCA)*, 2010.
- [9] S. Agarwal, R. Agarwal, B. Montazeri, M. Moshref, K. Elmeleegy, L. Rizzo, M. A. de Kruijf, G. Kumar, S. Ratnasamy, D. Culler, and A. Vahdat, “Understanding Host Interconnect Congestion,” in *ACM Workshop on Hot Topics in Networks (HotNets)*, 2022.
- [10] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, “Understanding PCIe performance for end host networking,” in *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.
- [11] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Brummer, and L. Van Doorn, “The Price of Safety: Evaluating IOMMU Performance,” in *The Ottawa Linux Symposium (OLS)*, 2007.
- [12] PCI-SIG, “PCI Express 6.0 Specification,” <https://pcisig.com/pci-express-6.0-specification>, 2022.
- [13] J. Hansbrough, “Fuzzing PCI express: security in plaintext,” <https://cloud.google.com/blog/products/gcp/fuzzing-pci-express-security-in-plaintext>, 2017.
- [14] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafir, “rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers,” in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.
- [15] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, “PAC it up: Towards pointer integrity using ARM pointer authentication,” in *USENIX Security Symposium (Usenix Security)*, 2019.
- [16] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, “PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.
- [17] S. Jero, N. Burow, B. Ward, R. Skowrya, R. Khazan, H. Shrobe, and H. Okhravi, “TAG: Tagged Architecture Guide,” *ACM Computing Surveys*, 2022.
- [18] H. Lefeuvre, N. Dautenhahn, D. Chisnall, and P. Olivier, “SoK: Software Compartmentalization,” in *IEEE Symposium on Security and Privacy (SP)*, 2025.
- [19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, 2011.
- [20] J. Cong, Z. Fang, M. Gill, and G. Reinman, “PARADE: A cycle-accurate full-system simulation Platform for Accelerator-Rich Architectural Design and Exploration,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015.
- [21] Xilinx, “Xilinx QEMU,” <https://github.com/Xilinx/qemu>, 2025.
- [22] —, “Xilinx systemctlm-cosim-demo,” <https://github.com/Xilinx/systemctlm-cosim-demo>, 2025.
- [23] Cadence, “Stratus High-Level Synthesis,” https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html, 2025.
- [24] “Moat,” <https://github.com/moat-cc>, 2026.
- [25] AMD, “AMD I/O Virtualization Technology (IOMMU) Specification (Rev. 3.10),” https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/specifications/48882_IOMMU.pdf, 2025.
- [26] Intel, “Intel Virtualization Technology for Directed I/O (VT-d),” <https://cdrdv2.intel.com/v1/dl/getContent/831418?fileName=vt-directed-io-spec.pdf>, 2025.
- [27] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. Moore, and R. Watson, “Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals,” in *Network and Distributed System Security Symposium (NDSS)*, 2019.
- [28] L. Yan, “ATS vs IOMMU-regroup - A Journey to Optimize GDRDMA in Cloud Scenarios,” <https://lpc.events/event/18/contributions/1685/attachments/1424/3047/ATSvsIOMMU-lyan.pdf>, 2024.
- [29] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, “Border control: Sandboxing accelerators,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.
- [30] F. Alam, H. Lee, A. Bhattacharjee, and A. Awad, “CryptoMMU: Enabling Scalable and Secure Access Control of Third-Party Accelerators,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.
- [31] M. Vuppapapati and R. Agarwal, “Tiered Memory Management: Access Latency is the Key!” in *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2024.
- [32] M. Bellare, “New Proofs for NMAC and HMAC: Security Without Collision-Resistance,” in *Advances in Cryptology (CRYPTO)*, 2006.
- [33] Canonical, “linux 5.4.0-64.72 source package in Ubuntu,” <https://launchpad.net/ubuntu/+source/linux/5.4.0-64.72>, 2021.
- [34] OpenSSL, “openssl,” <https://github.com/openssl/openssl>, 2025.
- [35] J. Cong, Z. Fang, M. Gill, and G. Reinman, “parade-ara-simulator,” <https://github.com/cdsc-github/parade-ara-simulator>, 2022.
- [36] QEMU, “QEMU TCG Plugins,” <https://www.qemu.org/docs/master/devel/tcg-plugins.html>, 2025.
- [37] M. Westerberg, “iommu/vt-d: Do not enable ATS for untrusted devices,” <https://github.com/torvalds/linux/commit/fb58fdcd295b914ec e1d829b24df00a17a9624bc>, 2018.
- [38] L. Baolu, “iommu/vt-d: Disable ATS support on untrusted devices,” <https://github.com/torvalds/linux/commit/d8b8591054575f3237556c32762d54e30774d28>, 2019.

- [39] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A Formal Foundation for Secure Remote Execution of Enclaves," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.
- [40] D. Lee, K. Cheang, A. Thomas, C. Lu, P. Gaddamadugu, A. Vahldiek-Oberwagner, M. Vij, D. Song, S. A. Seshia, and K. Asanovic, "Cerberus: A Formal Approach to Secure and Efficient Enclave Memory Sharing," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.
- [41] Z. Dai, S. Liu, V. Sjöberg, X. Li, Y. Chen, W. Wang, Y. Jia, S. N. Anderson, L. Elbeheiry, S. Sondhi *et al.*, "Verifying Rust Implementation of Page Tables in a Software Enclave Hypervisor," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.
- [42] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor," in *USENIX Security Symposium (Usenix Security)*, 2021.
- [43] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "Acai: Protecting Accelerator Execution with Arm Confidential Computing Architecture," in *USENIX Security Symposium (Usenix Security)*, 2024.
- [44] C. Wang, F. Zhang, Y. Deng, K. Leach, J. Cao, Z. Ning, S. Yan, and Z. He, "CAGE: Complementing Arm CCA with GPU Extensions," in *Network and Distributed System Security Symposium (NDSS)*, 2024.
- [45] F. Sang, J. Lee, X. Zhang, and T. Kim, "PORTAL: Fast and Secure Device Access with Arm CCA for Modern Arm Mobile System-on-Chips (SoCs)," in *IEEE Symposium on Security and Privacy (SP)*, 2025.
- [46] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao *et al.*, "StrongBox: A GPU TEE on Arm Endpoints," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.
- [47] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stappf, "CURE: A Security Architecture with Customizable and Resilient Enclaves," in *USENIX Security Symposium (Usenix Security)*, 2021.
- [48] CXL-Consortium, "CXL Specification," <https://www.computeexpresslink.org/download-the-specification>, 2025.