Towards (Really) Safe and Fast Confidential I/O

DOI:
https://doi.org/10.1145/3593856.3595913

Document Version
Final published version

Towards (Really) Safe and Fast Confidential I/O

Hugo Lefeuvre[†], David Chisnall[∞], Marios Kogias[‡∞], Pierre Olivier[†]
[†]The University of Manchester, [∞]Azure Research Microsoft, [‡]Imperial College London

ABSTRACT

Confidential cloud computing enables cloud tenants to distrust their service provider. Achieving confidential computing solutions that provide concrete security guarantees requires not only strong mechanisms, but also carefully designed software interfaces. In this paper, we make the observation that confidential I/O interfaces, caught in the tug-of-war between performance and security, fail to address both at a time when confronted to interface vulnerabilities and observability by the untrusted host. We discuss the problem of safe I/O interfaces in confidential computing, its implications and challenges, and devise research paths to achieve confidential I/O interfaces that are both safe and fast.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; • **Software and its engineering → Operating systems**.

ACM Reference Format:
Hugo Lefeuvre, David Chisnall, Marios Kogias, Pierre Olivier. 2023. Towards (Really) Safe and Fast Confidential I/O. In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3593856.3595913

1 INTRODUCTION

Confidential cloud computing [50] enables tenants to distrust their service provider. By encrypting data and manipulating it within a Trusted Execution Environment (TEE), confidential computing technologies can guarantee that even strong attackers (*e.g.,* insider threats) controlling the hypervisor/-cloud infrastructure remain unable to access tenant data. As a solution to the problem of data privacy [46] in public clouds, there is a very strong demand for confidential computing across the industry and regulatory bodies [18].

Achieving confidential computing solutions that provide concrete security guarantees requires 1) robust isolation, encryption and attestation mechanisms for enforcing the integrity and confidentiality of data throughout its lifetime and 2) carefully designed software interfaces between trusted and untrusted components. Both are critical: there is no confidentiality without strong mechanisms [38, 45, 66], but by themselves they are insufficient to protect a TEE in the presence of unsafe software interfaces [13, 33, 34, 61].

There are two dominant confidential computing paradigms: enclaves [16], that offer abstractions to run parts of an application inside the TEE; and confidential VMs [4, 32], that turn the popular VM abstraction confidential. These paradigms guarantee data confidentiality and integrity during compute. There are independent solutions covering similar guarantees for data at rest [56, 57] and in transit [6, 22, 47]. Yet, *the transitions between those states*, happening through I/O interfaces, remain challenging and potential attack vectors.

We make the following key observation: unlike TEE mechanisms, the topic of confidential I/O interface safety has been underexplored and even explicitly ignored in prior works, *e.g.,* rkt-io [55] and ShieldBox [58] use unhardened (or partially hardened) DPDK drivers for network communication. This is concerning: as we show in this paper, the design of safe and efficient I/O interfaces is particularly difficult and error-prone. Unfortunately, previous works aimed at TEE [61] and compartment [26, 30, 34, 41] interfaces is helpful in characterizing the problem but too broad or generic to provide solutions for efficient and safe I/O interfaces. This highlights a need for research in that domain.

This problem is particularly challenging because the design space is vast. Independently of the underlying mechanism (enclaves/confidential VMs), there are two main differences compared to prior work on interface safety [26, 30, 34, 61]. First, at the host end of the TEE there can be an application-specific interface providing I/O services, as implemented by different confidential computing frameworks [5, 9, 12, 21, 49, 52, 59]; a paravirtualized device; or a directly attached hardware device. Second, I/O depends on deep protocol stacks that allow for different separation of concerns between trusted/untrusted components, leading to different confidentiality guarantees, performance, and porting effort.

The status quo of confidential I/O interface design is not ideal. Intra-enclave library OSes/TEE frameworks [5, 12, 49, 52] limit the complexity of the interface with the untrusted host by internally managing as many system features as possible, but suffer from an increased TEE TCB size [3]. Further, I/O still needs to go through the host which leads to

HotOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595913

non-negligible performance losses [12, 52]. To address that issue, other systems offer direct hardware access to the enclave [7, 55, 58]. Not only does this solution not scale to large numbers of TEEs, it is also problematic in terms of security [29]: as we highlight in this paper, driver interfaces have neither been designed nor implemented for mutual distrust, and retrofitting distrust into them is hard. The issue is similar for confidential VMs, where traditional device drivers [20] lack the hardening necessary for confidential workloads [29]. In the case of confidential VMs, the problem is even more significant as they are widely publicized as lift-and-shift solutions reusing traditional driver stacks [20].

Coming up with a satisfying solution is hard because of the fundamental tug-of-war between security and performance: security benefits from simpler interfaces, copies, and checks, whereas performance benefits from lower-level, highly configurable interfaces with zero copies/checks.

In this paper we argue that securing confidential I/O interfaces requires to 1) carefully identify where the host/TEE trust boundary should be located in the I/O stack, and to 2) achieve a host/TEE interface that is safe by construction (as opposed to relying on ad-hoc checks). We discuss alternative boundaries across the I/O stack (§2.4), based on which we propose a ternary trust model between the confidential application, the I/O stack, and the device (§3.1). To achieve interfaces that are safe by construction, we advocate for alternative interfaces: analyzing the hardening effort in popular paravirtualized drivers (§2.5), we hint that hardening existing interfaces may not be the way to go for achieving safe and fast confidential I/O. Focusing on networking, we conclude providing design guidelines to achieve safe confidential I/O and reach different performance, compatibility, and confidentiality trade-offs (§3.2).

2 CHALLENGES OF CONFIDENTIAL I/O INTERFACES

2.1 Trust Model

Confidential Workload. The confidential workload (Ⓣ in Figure 1) is composed of an application (or part of it), and of a confidential computing framework. The nature of the framework depends on the type of TEE: in the case of enclaves, the framework wraps interactions with the untrusted kernel Ⓢ and other parts of the application, while in the case of confidential VMs, the framework is the trusted OS that runs on top of the untrusted hypervisor Ⓢ. All elements of Ⓣ are trusted in the general case, although, as we describe in §3, more fine-grained trust relationships can be achieved with compartmentalization. *We aim to protect Ⓣ's confidentiality and integrity.* We consider Denial of Service (DoS) out of scope, as TEE mechanisms themselves do not support it. We consider architectural side-channels [11, 24, 28, 60, 67] and mechanism-related vulnerabilities [38, 45, 66] out of scope, as mitigations are orthogonal to our aims.

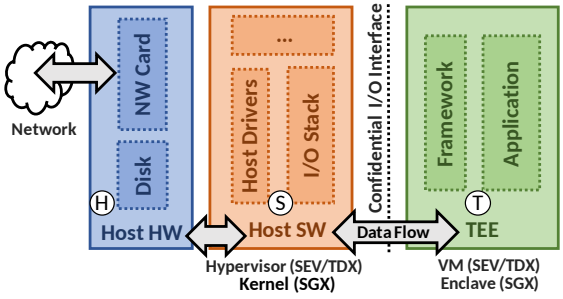

Figure 1: Key components of confidential computing architectures and confidential I/O boundary.

Host Software. The host software Ⓢ has a mutual distrust relationship with Ⓣ; *i.e.,* Ⓢ does not trust Ⓣ and Ⓣ does not trust Ⓢ. In the general case, Ⓢ includes an I/O stack and drivers to interface with the host hardware Ⓗ. In the case of enclaves, Ⓢ is an operating system, while in the case of confidential VMs it is a hypervisor. The boundary between Ⓣ and Ⓢ includes the confidential I/O interface.

Host Hardware. The host hardware Ⓗ is distrusted by Ⓣ in the general case. It includes networking hardware, disks, etc. In the case of direct device assignment, as we discuss in §3.4, part of Ⓗ may be trusted (or partly trusted) by Ⓣ.

2.2 Ideal Confidential I/O Properties

Ideal: Interface Safety by Construction. Confidential I/O designs must guarantee the confidentiality and integrity of the confidential workload by construction. This means designing with two main vulnerability vectors in mind:

- *Interface vulnerabilities*: the design of the I/O boundary must avoid or at least minimize vulnerabilities triggered through interface misuses [13, 33, 34, 61], leading to information leakage, data corruption, as well as temporal violations [34], at the exposed protocol layers.
- *Observability by the host*: the design of the I/O boundary must minimize the amount of non-architectural side-channels exposed to the host (*e.g.,* I/O metadata, ordering and types of I/O calls), as these allow the host to infer information about the TEE [3].

We develop on these two vectors in the following sections. Unlike traditional I/O interfaces, the host is not trusted, making it even more challenging to achieve high performance and strong security properties.

Ideal: High Performance. Confidential I/O designs must be capable of handling high throughput and low latencies. For example, with networking this means saturating a link,

i.e., tens of Gbit/s [17, 58]. For storage, this also means throughputs of 10+ Gbit/s [27].

Ideal: Independence from TEE Implementation. I/O designs should be able to plug with the different scenarios highlighted by §2.1. I/O services can be provided by an OS kernel as well as by a hypervisor, leveraging a range of technologies (*e.g.,* enclaves, confidential VMs).

Relaxed: Backwards Compatibility. Confidential workloads exhibit generally relaxed backwards compatibility requirements. The cloud infrastructure can be adapted to cater for safer confidential workloads: major cloud players are members of the Confidential Computing Consortium [15]. This makes it reasonable to change the hypervisor, or even push towards hardware changes. For application- and OS-code, compatibility is generally desirable [12, 39] but reasonable to trade-off for security and/or performance, ideally in an iterative manner. There is general awareness that getting the most out of TEEs cannot be achieved without changes across the stack [50, 61], and the community has shown willingness to perform these changes [49].

2.3 Confidential I/O: Divide and Rule

Given the previously-mentioned requirements, we propose to divide the problem in two parts: 1) at what abstraction level to place the I/O trust boundary between Ⓣ and Ⓢ, and 2) how to design the I/O interface at the selected level.

P1: Where to Position the I/O Trust Boundary? The trust boundary between the host and the TEE can be placed at various levels in the I/O stack. This influences the nature of the data flowing through the interface. With networking, for instance, data can be raw packets from the network, a TLS-encrypted TCP flow, raw UDP packets, etc. For storage, it can be filesystem operations, block I/Os, etc. Placing the trust boundary at different levels will expose different trade-offs influencing both interface vulnerabilities and observability: important metrics impacted will be the size of the TCB in the confidential domain (*e.g.,* does it contain or not a large TCP/IP stack or a filesystem [3]), as well as the complexity of the driver (*e.g.,* if we get rid of packet layers, we can shave off sources of complexity like negotiation of MTU, who handles the checksum, MAC address, etc.). Having control over the overall design of the interface, *we have the ability to choose at what protocol level we want to position the trust boundary.*

P2: How to Design the I/O Interface Itself? The design of the I/O interface itself defines how data is exchanged over the interface. Many I/O interface designs have been proposed over the years; MMIO-exposed registers, ring buffers, indirect descriptors, various approaches for zero-copy I/O, etc. For example, different approaches may or may not share pointers and/or indexes in safe manners, *e.g.,* via different numbers of indirection layers, resulting in more or less resilience to interface vulnerabilities [34]. Beyond this, different approaches show different control/data path complexity, resulting in varying degrees of statefulness and complexity of error paths, all again impacting interface resilience [34].

We now go through these two problems in §2.4 (P1) and 2.5 (P2), highlighting limitations of previous works and motivating the research paths we propose in §3.

2.4 P1: I/O Trust Boundary Location

A range of approaches have been explored regarding P1. Enclave approaches that perform networking via the system call interface [5, 12, 49, 52] operate at OSI layer 5 (L5, TCP/UDP flows), as exposed by the OS's socket abstraction. However, almost all high-performance approaches [20, 44, 55, 58] work at layer 2 (L2), exchanging raw Ethernet packets, processed by the TEE's own I/O stack. Some works [17] also explore tunneled approaches, encapsulating L2 packets into a TLS tunnel from a safe network, to hide metadata from the confidential unit's untrusted host and network.

There are valid reasons for most works to focus on layer 2. First, high-performance approaches build on device pass-through (enclaves), or high-performance paravirtualized devices (confidential VMs), which both require raw Ethernet packets. Second, operating at higher layers means that more observability [3] is exposed to the host, such as the timings of `accept`, or the socket configuration options. At layer 2, the amount of observability is equivalent to what one could obtain by simply observing the network.

On the other hand, there are also arguments to place the boundary higher up at layer 5. Even though the network subsystem (TCP/IP stack, drivers) has been hardened against network threats over the years, it is ever-growing (about +20% LoC per major version), and remains widely affected by remotely-exploitable vulnerabilities (Figure 2). Placing it within the confidential workload's TCB is concerning and a clear violation of the principle of least privilege. Assuming a secure interface design (P2), if layer 2 is chosen for P1, it is on the I/O stack that attackers will focus, and its compromise will jeopardize the efforts deployed on securing the I/O boundary. This also provides strong arguments towards a confidential I/O boundary at layer 5, assuming suitable performance can be achieved.

Both layers 2 and 5 are suitable to position the I/O boundary, as the interfaces can be hardened. This is not the case of layer 6, above TLS: although similar argumentation on vulnerability/size could be made for TLS, assuming an untrusted TLS component negates the confidentiality of data and opens

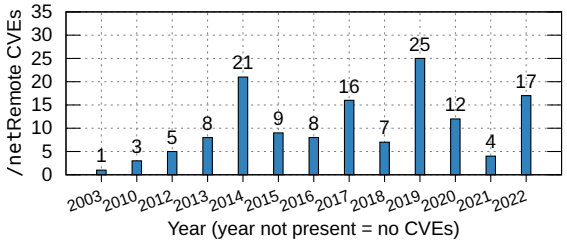

Figure 2: Remotely-exploitable CVEs in the Linux `/net` subsystem over the years.

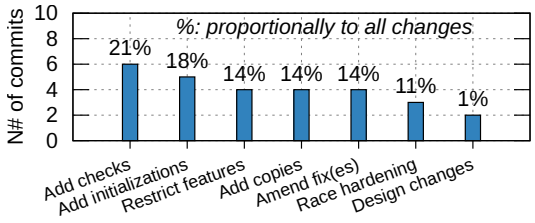

Figure 3: Distribution of hardening commits to the Linux paravirtualized networking `netvsc` driver.

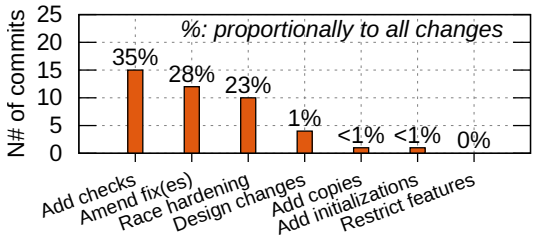

Figure 4: Distribution of hardening commits to the Linux `virtio` paravirtualized driver family.

for numerous interface vulnerabilities, as the ordering and integrity of payloads is not guaranteed anymore.

We focused here on networking for the sake of brevity, but most observations also apply to storage that similarly has high-level (*e.g.,* filesystem operations) and low-level (*e.g.,* block I/Os) components. Overall, there is no confidential I/O stack approach that combines low TCB (*e.g.,* layer 5 boundary), low observability (*e.g.,* layer 2 boundary), and high performance (rather bound to layer 2); we sketch an approach that relies on a dual-boundary system in §3.

2.5 P2: Designing a Secure I/O Interface

P2 clearly sparks two alternative resolution approaches: hardening existing I/O interfaces, or defining new ones.

Regarding hardening existing interfaces, we observe ongoing efforts on hardening paravirtualized device driver implementations in Linux [43, 64]. These efforts aim to retrofit mutual distrust into paravirtual I/O device drivers, while remaining compliant to the standards (which were *not* designed with distrust in mind). As we discuss here, this approach is fundamentally limited by the need for backwards compatibility, and by the broad scoping of the standards, ultimately impacting security and performance.

In order to understand the implications and benefits of hardening existing drivers, and identify characteristics that should be focused on in a from-scratch approach, we systematically record all ongoing hardening efforts performed on the VirtIO [48] and NetVSC [62] Linux paravirtual device drivers. We classify each (merged) commit according to the type of change performed. We record 7 types of changes: adding checks, adding initialization to memory, adding copies, protecting against races, restricting features, performing design changes, or amending previous hardening commits. Figures 3 and 4 show the distribution of commits by category[1].

We make four key observations. First, hardening is extremely error-prone. In the VirtIO case for example, over 40 commits, 12 either revert or amend previous hardening changes, some of them never to be re-applied. Second, much of this complexity is coming from the need to support legacy compatibility features and implementation behaviors. In several cases this need for backwards compatibility prevails, causing the revert of hardening features. Third, the complexity also stems from the large scope of the paravirtual standards. For instance, VirtIO strives to accommodate both paravirtual scenarios and hardware implementations all the same. This results in increased complexity on the control (or configuration) path, with extensive, stateful configuration protocols that open for non-trivial timing and ordering vulnerabilities [34]. The VirtIO standard for example supports at least two alternative *virtqueue* (data transport mechanism) formats, each featuring unique hardening needs. Finally, performance tends to suffer from the hardening more than needed: this is due to disabling security features that bring performance benefits (*e.g.,* in NetVSC), or piggybacking copies on the protocol when the latter wasn't thought with them in mind (SWIOTLB [36] in Linux, applied to VirtIO and NetVSC, which copies systematically even in cases where double fetch is impossible). This increased cost is hard to avoid when retrofitting distrust within the frame of a standard that does not mandate it.

Since they are structural and bound to the designs and motivations of standards, these observations also apply to unhardened driver implementations such as DPDK or SPDK, both popular among enclaves [7, 44, 55, 58]. Other works also explore from-scratch approaches [17], but none of them with a focus on interface vulnerabilities; as a result, even these

[1]The raw data for Figures 2 to 4, along with relevant scripts, is available open-source at https://github.com/hlef/cio-hotos23-data.

are prone to similar observations. Overall, no existing stack tackles interface vulnerabilities by design while maintaining performance. There is a need for more research to come up with principled I/O interfaces that, by conception, are resilient against interface threats, which we sketch in §3.

3 MAKING CONFIDENTIAL I/O RIGHT

We propose a novel confidential I/O design that addresses the limitations identified in the previous section. Different I/O types may require different boundary design decisions, thus for space reasons this section focuses on networking. We discuss a generalization of our principles in §3.3.

3.1 P1: Plugging at the Right Interface Level

We identify two candidate layers to position the confidential I/O boundary: L2 (raw Ethernet packet) and L5 (TLS-encrypted TCP flow). As discussed previously, both can reasonably be hardened or designed with safety in mind, but neither is ideal: L2 implies a large TCB (I/O stack) in the confidential unit, and L5 results in richer observability by the host, among others.

To address that issue, we propose a *dual-boundary* approach that yields the best of both worlds: we explore an API design combining a strong boundary at L2, limiting observability, and a lightweight one at L5, excluding the I/O stack from the core TCB. This results in a ternary/nested trust model: the set of the I/O stack and the rest of the confidential unit (including the confidential application) does not trust the outside world (host), *yet the I/O stack is not trusted by the rest of the confidential unit either*. The result is an approach that combines 1) low observability, since a powerful attacker on the host does not have access to more information than it would by monitoring the network; with 2) a significantly lower TCB, raising the bar to compromise the confidential workload: compromising the I/O stack, whether through protocol or interface vulnerabilities, only results in increased observability. The host must now mount multi-stage attacks to compromise the confidential application. We represent this intermediate area in Figure 5, alongside other solutions.

Many design approaches can be taken to enforce the dual boundary. The I/O stack and the rest of the confidential unit could run in two separate TEEs (*e.g.,* two enclaves); however such an additional heavyweight protection domain switch on the I/O path would unnecessarily hurt latency by introducing a dual distrust boundary at L5 where only *single* distrust is needed, as the I/O stack trusts the rest of the confidential unit. Thus, a compartment-based approach [35, 41, 65] relying on low-latency memory isolation techniques within the TEE [25, 51, 52] is more appropriate. In this case, the L2 boundary is a strong host-TEE boundary, and the L5 boundary is a lightweight intra-TEE compartment boundary. Performance is preserved, and security is enhanced compared to an approach without protection at L5.

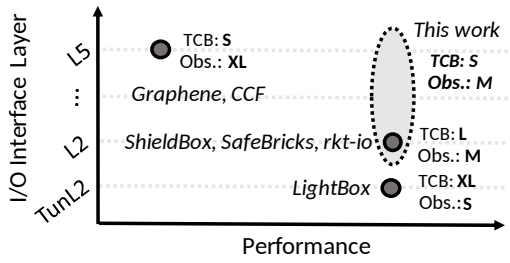

Figure 5: Leverage control over P1 to reduce TCB and observability (Obs.) without hurting performance.

3.2 P2: Achieving Strong I/O Boundaries

Hardening L2. In order to achieve a host-TEE boundary at L2 that is by conception resilient to interface vulnerabilities, we propose a design based on five main principles:

- *Stateless Interface*: there are no dependency relationships across and within data/control plane operations. This considerably simplifies safety reasoning. This means, among others, eliminating error paths whenever possible: in a networked confidential workload for instance, failure to `bind()` likely should be fatal.
- *Copy as a first-class citizen.* Copies are part of the protocol: they are performed early, but only when necessary, and avoided when possible (*e.g.,* when double fetches are absent by design). This improves over legacy approaches where the copy is typically piggybacked everywhere [36].
- *No notifications*: these introduce concurrency that is difficult to protect (Figure 4), and do not contribute to performance under polling scenarios. When polling is not an option, this must be relaxed; notification handlers are then designed stateless, idempotent, and thread-safe.
- *Zero (re-)negotiation*: parameters like MAC address, MTU size, or who calculates checksums are known at device startup and can be fixed during deployment. This contributes to a minimal control plane. This does not fundamentally preclude live migration, as devices can be hot-swapped; nevertheless, migration without downtime [63] remains difficult as it introduces statefulness in the protocol that is difficult to secure.
- *Safe ring buffer & shared data area*: by construction, pointers to the ring buffer and shared-data area must be protected via careful pointer/index masking [14]: the size of shared memory areas and buffers along with their alignment is designed to make these operations efficient/possible (*e.g.,* alignement at a power of two).

Further, we explore avenues to maintain high-performance.

- *Explore data positioning.* Several designs should be explored: inlining data on the ring buffer together with descriptors, separately in shared memory with mask-protected pointers, or mask-protected indirect descriptors. If buffers are stored separately, buffer freeing should be safe by conception, *e.g.,* via control messages, or with a host-TEE shared memory allocator designed for distrust [40].
- *Explore revocation.* On the receiving side, we explore avoiding copies safely by un-sharing pages with the host on the fly; *we explore when this becomes faster than copies, and how to integrate it harmoniously in interface semantics.*

Hardening L5. We design L5 to be resilient by design to interface vulnerabilities based on the following principles:

- *Avoid the need to verify pointers.* Leverage the single distrust to enforce a *trusted component allocates* policy [34]: the application allocates directly in the I/O domain when sending, and provides the buffer when receiving.
- A mandatory TLS layer guarantees data integrity and confidentiality, notably against attempts to break TCP guarantees (*e.g.,* replay attacks, out of order packets, etc.).

Here too we explore a range of performance design choices:

- *How can we achieve zero-copy send on the confidential side?* Here too, leverage the single distrust relationship between the I/O stack and the confidential workload.
- On the receiving side we may need a copy at L5 if we do not trust the I/O stack. Here too we propose to *explore revocation*, as discussed for L2, to eliminate that copy.

3.3 Discussion: Beyond Networking

The two-boundary solution to P1 (§3.1) should map well to other I/O boundaries that also have observability problems, *e.g.,* storage [3]. Here, the first boundary would be at a low-level interface, *e.g.,* disk driver or block layer, and the second one at a higher level such as file operations. Our approach to P2 (§3.2) at both layers should also transfer well to other types of I/O. However, it is likely that it will not always be possible to rewrite drivers at the lowest level (e.g, GPU). In this case, rethinking the answer to P1 should be considered: positioning the boundary just after the driver by compartmentalizing it may be a valid solution as well. Compartmentalizing is realistic: it has been advocated to secure drivers for many years [10, 37, 42, 54] and recently it is getting increasingly automated [31].

3.4 Discussion: Direct Device Assignment

Even though Direct Device Assignment (DDA) suffers from the same problems, *i.e.,* the device itself can be malicious, and the communication channel between the TEE and the device is exposed to a malicious host, the hardware community has taken a different path in dealing with the problem: extending PCIe, used to interconnect I/O devices, with support for secure communication and device attestation. Given that the TEE can attest the device, it can trust it/add it to its TCB. Also, given that the communication channel between the TEE and the attested device is encrypted/integrity-protected, there is no need to harden drivers.

Specifically, the TEE Device Interface Security Protocol describes an architecture for confidential communication between TEEs and PCI-attached devices. Here, PCIe communications are encrypted using IDE (Integrity & Data Encryption) and there is an attestation protocol between the TEE and device according to SPDM [1] (Security Protocol and Data Model). Different TEE implementations need to implement those PCIe specs, *e.g.,* Intel's TDX [2], through a combination of software and hardware mechanisms.

Despite the performance benefits of directly attached devices, there are limitations in the granularity of partitioning them; DDA is not a silver-bullet, especially with high oversubscription, which paravirtual devices can tackle. Security-wise, DDA is not perfect either: even trusted/attested devices can be compromised (particularly as their complexity is increasing [19]), and adding them to the trusted TCB is a trade-off by itself. SR-IOV-specific attacks have also been described in the past [8, 23, 53, 68, 69], although they are generally limited to availability threats [23, 53, 69] and covert channels [8].

4 CONCLUSION

We highlighted the challenging and often ignored problem of I/O interface safety for TEEs. Studying the hardening effort in widely used systems by the open source community, we show that hardening existing interfaces can only lead to a dead end given the mismatch between the threat model in confidential computing and the threat model in traditional virtualization. Instead, we propose domain-specific interfaces that are easier to harden, can offer both confidential and high-performance I/O, and can be implemented through a ternary trust model between the confidential application, the I/O stack, and the paravirtual device.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insights. We are also grateful to Jason Wang and Istvan Haller for their insightful feedback. This work was partly funded by a studentship from NEC Labs Europe, a Microsoft Research PhD Fellowship, the UK's EPSRC grants EP/V012134/1 (UniFaaS), EP/V000225/1 (SCorCH), and the EPSRC/Innovate UK grant EP/X015610/1 (FlexCap).

REFERENCES

[1] [n.d.]. Security Protocol and Data Model (SPDM) Specification. https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.2.1.pdf Accessed Jan, 28 2023.

[2] [n.d.]. Software Enabling for Intel® TDX in Support of TEE-I/O. https://cdrdv2.intel.com/v1/dl/getContent/742542 Accessed Jan, 28 2023.

[3] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX.. In *Proceedings of the 25th Annual Network & Distributed System Security Symposium (NDSS'18)*.

[4] AMD Inc. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf Accessed Dec, 19 2022.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure linux containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[6] Atkinson, R. 1995. Security Architecture for the Internet Protocol. https://www.rfc-editor.org/rfc/rfc1825 Accessed Jan, 28 2023.

[7] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*.

[8] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. 2014. On detecting co-resident cloud instances using network flow watermarking techniques. *International Journal of Information Security* 13 (2014).

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3. https://doi.org/10.1145/2799647

[10] Silas Boyd-Wickizer and Nickolai Zeldovich. 2010. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Conference on Offensive Technologies (WOOT'17)*.

[12] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[13] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. https://doi.org/10.1145/2451116.2451145

[14] David Chisnall. 2008. The Definitive Guide to the Xen Hypervisor (Section 6.3). https://www.informit.com/articles/article.aspx?p=1160234&seqNum=3 Accessed Feb, 2nd 2023.

[15] Confidential Computing Consortium. 2023. Confidential Computing Consortium Members. https://confidentialcomputing.io/members/ Accessed Jan, 30 2023.

[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016). https://eprint.iacr.org/2016/086

[17] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. 2019. LightBox: Full-Stack Protected Stateful Middlebox at Lightning Speed. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. https://doi.org/10.1145/3319535.3339814

[18] Everest Global, Inc. 2021. Confidential Computing - The Next Frontier in Data Security. https://confidentialcomputing.io/wp-content/uploads/sites/85/2021/10/Everest_Group_-_Confidential_Computing_-_The_Next_Frontier_in_Data_Security_-_2021-10-19.pdf Accessed Dec, 20 2022.

[19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*.

[20] Xinyang Ge, Hsuan-Chi Kuo, and Weidong Cui. 2022. Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*. https://doi.org/10.1145/3548606.3560592

[21] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-based Construction of Trusted Execution Environments. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*. http://www.usenix.org/conference/atc19/presentation/ghosn

[22] Google, Inc. 2022. Encryption in transit (Whitepaper). https://cloud.google.com/docs/security/encryption-in-transit Accessed Jan, 28 2023.

[23] Yonatan Gottesman and Yoav Etsion. 2016. NeSC: Self-virtualizing nested storage controller. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*.

[24] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. https://doi.org/10.1145/3065913.3065915

[25] Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. A Hardware-Software Co-design for Efficient Intra-Enclave Isolation. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security'22)*. https://www.usenix.org/conference/usenixsecurity22/presentation/gu-jinyu

[26] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. https://doi.org/10.1145/2810103.2813611

[27] Gygabyte. 2022. GIGABYTE Leads and Reveals the First PCIe 5.0 SSD. https://www.gigabyte.com/Press/News/2017

[28] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'17)*.

[29] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. 2021. VIA: Analyzing Device Interfaces of Protected Virtual Machines. In *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC '21)*. https://doi.org/10.1145/3485832.3488011

[30] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. 2015. Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS'15)*.

[31] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2022. KSplit: Automating Device Driver Isolation. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. https://www.usenix.org/conference/osdi22/presentation/huang-yongzhe

[32] Intel Corporation. 2021. Intel(R) Trust Domain Extensions White Paper. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html Accessed Dec, 19 2022.

[33] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[34] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2023. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In *Proceedings of the 30th Annual Network & Distributed System Security Symposium (NDSS'23)*.

[35] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. https://doi.org/10.1145/3503222.3507759

[36] Tom Lendacky. 2017. x86/mm: Add DMA support for SEV memory encryption. https://github.com/torvalds/linux/commit/d7b417fa08d1187923c270bc33a3555c2fcff8b9 Accessed Feb, 1st 2023.

[37] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. 2004. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*.

[38] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD Secure Encrypted Virtualization. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*.

[39] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'17)*.

[40] Microsoft Corporation. 2023. snmalloc: Message passing based allocator. https://github.com/microsoft/snmalloc/tree/main/docs/security Accessed Feb, 1st 2023.

[41] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[42] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An Operating System with Kernel Virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. https://doi.org/10.1145/2517349.2522719

[43] Andrea Parri. 2022. hv_netvsc: Add validation for untrusted Hyper-V values. https://lkml.org/lkml/2020/9/16/380 Accessed Jan, 30 2023.

[44] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*.

[45] Martin Radev and Mathias Morbitzer. 2021. Exploiting Interfaces of Secure Encrypted Virtual Machines. In *Proceedings of the 4th Reversing and Offensive-oriented Trends Symposium (ROOTS'20)*. https://doi.org/10.1145/3433667.3433668

[46] Kui Ren, Cong Wang, and Qian Wang. 2012. Security Challenges for the Public Cloud. *IEEE Internet Computing* 16, 1 (2012). https://doi.org/10.1109/MIC.2012.14

[47] Rescorla, E. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. https://www.rfc-editor.org/rfc/rfc8446 Accessed Jan, 28 2023.

[48] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[49] Mark Russinovich et al. 2019. *CCF: A Framework for Building Confidential Verifiable Replicated Services.* Technical Report. Microsoft Research and Microsoft Azure. https://github.com/microsoft/CCF/blob/main/CCF-TECHNICAL-REPORT.pdf

[50] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. 2021. Toward Confidential Cloud Computing. *Commun. ACM* 64, 6 (2021). https://doi.org/10.1145/3453930

[51] Vasily A. Sartakov, Daniel O'Keeffe, David Eyers, Lluís Vilanova, and Peter Pietzuch. 2021. Spons & Shields: Practical Isolation for Trusted Execution. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'21)*. https://doi.org/10.1145/3453933.3454024

[52] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. https://doi.org/10.1145/3373376.3378469

[53] Igor Smolyar, Muli Ben-Yehuda, and Dan Tsafrir. 2015. Securing Self-Virtualizing Ethernet Devices. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/smolyar

[54] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. 2002. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th ACM SIGOPS European Workshop (EW'10)*. https://doi.org/10.1145/1133373.1133393

[55] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. 2021. Rkt-Io: A Direct I/O Stack for Shielded Execution. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys'21)*. https://doi.org/10.1145/3447786.3456255

[56] The kernel development community. 2023. dm-crypt: Linux kernel device-mapper crypto target. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-crypt.html Accessed Jan, 28 2023.

[57] The kernel development community. 2023. Filesystem-level encryption (fscrypt). https://www.kernel.org/doc/html/latest/filesystems/fscrypt.html Accessed Jan, 28 2023.

[58] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2018. ShieldBox: Secure Middleboxes Using Shielded Execution. In *Proceedings of the 2018 Symposium on SDN Research (SOSR'18)*. https://doi.org/10.1145/3185467.3185469

[59] Chia-che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. https://www.usenix.org/conference/usenixsecurity20/presentation/tsai

[60] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient out-of-Order Execution. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security'18)*.

[61] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. https://doi.org/10.1145/3319535.3363206

[62] Anthony Velte and Toby Velte. 2009. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc.

[63] Laurent Vivier. 2022. Virtio-net failover: An introduction. https://www.redhat.com/en/blog/virtio-net-failover-introduction Accessed Jan, 30 2023.

[64] Jason Wang and Ariel Adam. 2022. Hardening Virtio for emerging security use cases. https://www.redhat.com/en/blog/hardening-virtio-emerging-security-usecases Accessed Jan, 30 2023.

[65] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy (S&P'15)*.

[66] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *2020 IEEE Symposium on Security and Privacy (S&P'20)*. https://doi.org/10.1109/SP40000.2020.00080

[67] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy (S&P'15)*. https://doi.org/10.1109/SP.2015.45

[68] Tao Zhang, Boris Pismenny, Donald E. Porter, Dan Tsafrir, and Aviad Zuck. 2021. Rowhammering Storage Devices. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21)*. https://doi.org/10.1145/3465332.3470871

[69] Zhe Zhou, Zhou Li, and Kehuan Zhang. 2017. All Your VMs Are Disconnected: Attacking Hardware Virtualized Network. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY'17)*. https://doi.org/10.1145/3029806.3029810