Conditional Network Availability: Enhancing Connectivity Guarantees for TEE-Based Services

Jonas Röckl
FAU Erlangen-Nürnberg
jonas.roeckl@fau.de

Christian Lindenmeier
FAU Erlangen-Nürnberg
christian.lindenmeier@fau.de

Matti Schulze
FAU Erlangen-Nürnberg
matti.schulze@fau.de

Tilo Müller
Hof University of Applied Sciences
tilo.mueller@hof-university.de

Abstract—**Trusted Execution Environments (TEEs) are widely available, allowing the isolation of security-sensitive trusted services from an untrusted commodity OS. Driven by manifold use cases, more and more trusted services requiring network connectivity are developed. Typically, the traffic of trusted services is routed through the OS, while cryptography ensures confidentiality and integrity. However, the extent to which TEEs can also help to provide network availability for trusted services remains underexplored.**

We introduce *Conditional Network Availability (CNA)* as a novel concept for TEE-based networking, ensuring that a trusted service can process network traffic, *whenever* the potentially malicious OS can do so. Our concept prevents an attacker from monopolizing the network channel (e.g., for a botnet campaign). TEE-based remote device management, system monitoring, and intrusion detection systems can profit from our concept.

Proposing a split-driver model, we implement a proof-of-concept on real hardware, multiplexing a complex Ethernet interface between the OS and the ARM TrustZone TEE. Our evaluation shows that our system achieves near-native throughput while keeping the additions to the TCB small.

1. Motivation

Trusted Execution Environments (TEEs) provide effective protection against application- and even kernel-level attackers. Driven by manifold use cases, TEE-based services are becoming increasingly complex [7]. Notably, many of the services (e.g., mobile device management [37], system monitoring [19], and control flow attestation [1], [34]) require network connectivity.

Naively, one approach to achieve network connectivity for trusted services is to migrate the *complete* network driver and its dependencies to the TEE and expose a virtual Network Interface Card (NIC) to the Rich Execution Environment (REE). While this method is well suited for hypervisors [30], it falls short for a TEE due to the large additions to the Trusted Computing Base (TCB) [4]. NIC drivers can span tens of thousands of Lines of Code (LoC) [27]. Therefore, TEE-based systems typically encrypt the payload before forwarding it to an agent within the REE [14]. This agent then assembles a network packet and forwards it relying on the network stack of the untrusted OS. While cryptography can provide integrity, confidentiality, and authenticity, there is no availability. An attacker can defer, replay, and block packets.

Traditionally, this has not posed an issue, since trusted services only activate when requested by the OS [39]. There was no computational availability for trusted services since the TEE depends on the REE scheduler to allocate CPU time for the TEE. However, there has been a growing interest in implementing computational availability for TEEs, meaning that the architecture guarantees that the TEE is scheduled regularly [3], [17], [24], [26], [29], [39], [40]. A common strategy is to use timers, which can only be controlled by the TEE and regularly interrupt the CPU to execute the TEE [3], [6], [12], [24], [39].

We explore how the computational availability of TEEs influences the network I/O of trusted services. In the networking domain, true availability can only be achieved through (physical) redundancy, if at all. Yet, we are reluctant to accept that the only practical approach is to rely on the REE to forward traffic, giving up any availability to the extent that an attacker can monopolize the network channel (e.g., for botnet campaigns [4], [35]).

Contributions. We aim to decrease the dependency of the TEE on the REE when conducting network I/O in trusted services. To that end, we develop a novel approach to multiplex a complex Ethernet interface between the TEE and the REE, while keeping a low TCB and ensuring certain network availability guarantees for trusted services. More precisely, we make the following contributions:

- We define the concept of Conditional Network Availability (CNA) as a property for TEE-based networking. In a nutshell, a TEE network interface complies with CNA if a trusted service can send and receive data whenever the REE can do so.
- We derive a system architecture, based on ARM TrustZone, capable of multiplexing an Ethernet interface between the TEE and the REE. We show that the interface is compliant with CNA.
- We create a proof-of-concept[1] on real, unmodified Commercial-Off-The-Shelf (COTS) hardware and measure network throughput and latency.
- We show that the CNA-compliant Ethernet interface for the TEE comes with small additions to the TCB and discuss security implications.

We focus on a CNA-compliant Ethernet interface for the TEE. Adding support for further protocols to the TEE, such as TCP/IP, is future work. However, these do not require peripheral access and can be isolated using existing techniques (Sec. 7).

1. https://github.com/conditional-network-availability/src

2. Background

ARM TrustZone. With ARM TrustZone, the device is partitioned into two distinct execution environments, the Normal World (NW), as a REE, with a full-fledged OS such as GNU/Linux, and the Secure World (SW), as a TEE, with stripped-down system software (referred to as SW OS). The *secure monitor* runs at the highest privilege level, handling context switches from and to the SW. The secure monitor is considered part of the TEE. To initiate a world switch, a Secure Monitor Call (SMC) is executed, triggering a synchronous exception that is taken to the secure monitor. At system startup, the secure monitor is launched first, followed by the initialization of the SW OS. Only after that, the NW starts. To prevent boot chain attacks, the authenticity of all SW software is verified during load time (secure boot). The Root of Trust (RoT), i.e., a public key to verify the signatures, must be protected from modifications. For example, some boards support burning a key into the hardware with eFuses. To alter or extend the behavior of TrustZone-based systems, one can either modify the SW OS or the secure monitor. We extend the secure monitor to provide a CNA-compliant network interface to the SW.

ARM TrustZone I/O. ARMv8-A uses Memory-Mapped I/O (MMIO) to communicate with peripherals. With a TrustZone Protection Controller (TZPC) or similar manufacturer-specific hardware, the TrustZone allows assigning peripherals to (exactly) one world, restricting access to their MMIO registers to that world. However, typical SW OSs, such as OP-TEE [22], do not access the NIC directly but forward packets to an agent in the NW.

Networking Drivers. From a broader perspective, an Ethernet NIC in interrupt mode works as follows. The NIC features configuration registers accessible via MMIO. By writing to them, the OS can set up an Ethernet connection. Ethernet NICs are centered around the concept of (descriptor) rings. A descriptor ring is a cyclic buffer in RAM, which contains (buffer) descriptors. A buffer descriptor is a data structure with a pointer to a frame in RAM and fields that describe the frame's state (e.g., the length of the frame and whether it is ready for transmission). The NIC accesses the descriptor rings via Direct Memory Access (DMA). There are MMIO configuration registers, written to by the CPU, that hold the address of each ring in RAM.

To transmit (TX) a frame, the OS fills a descriptor in a TX ring and marks it as ready. Subsequently, the NIC sends the frame. The position in the rings moves ahead. The NIC enriches the descriptor with status information. For general-purpose hardware, the OS sets up the NIC to issue an Interrupt Request (IRQ) after a transmission. Handling the IRQ, the CPU retrieves the status.

To receive (RX) frames, the OS prepares a RX ring pointing to empty buffers. When the NIC receives a frame, the NIC copies it to the next free buffer pointed to by a descriptor and sets a flag in the descriptor. Subsequently, the NIC issues an IRQ and the driver iterates through the new descriptors in the ring to process the frames.

There are multiple TX rings and RX rings. While NICs typically place all received frames in one RX ring, some models allow directing frames to a ring based on the frame's headers. This allows, for example, to implement Quality of Service (QoS) mechanisms [27].

3. Adversary Model

We focus on an attacker who compromises the NW OS. The attacker can modify kernel data structures, including the ones of the NIC driver in the NW. The attacker can also manipulate clocks, disable the NIC, or shut down the system. Our TCB is all the software in the SW, which is common for TEE-based systems. True network availability can only be attained through redundancy, if possible at all. Thus, we exclude safety-critical systems that rely on the timely delivery of packets.

We consider an attacker who communicates with the device over the network and exploits vulnerabilities in the NW to take over the device. We assume that attackers tend to maintain network connectivity to continuously control the device [37]. We do not address an attacker controlling the network itself, such as in man-in-the-middle attacks. Like Denial of Service (DoS) attacks (Sec. 7), such threats can only be mitigated at the infrastructure level.

We assume that the hardware and the TEE work as specified. We assume a secure boot chain and a fixed RoT. Side-channel attacks and physical attacks are out of scope.

4. Conditional Network Availability

We observe that trusted services frequently involve either the need to transmit (e.g., for system monitoring [8], [19]) or the necessity to receive data (e.g., commands for mobile device management [17], [31], [37] or software updates [10], [29], [40]). Relying on a network agent in the NW (Sec. 1), all of these systems are exposed to an attacker that monopolizes the network for their purposes (e.g., DDoS [4], [16]) while blocking TEE traffic. This becomes especially problematic if the attacker can trigger unfavorable system states this way (e.g., deferring the installation of security updates). To prevent an attacker from monopolizing the network channel, we define the concept of CNA as a property for a TEE network interface.

Definition. A TEE network interface complies with *Conditional Network Availability (CNA)* if there is assurance that the TEE can process data via the interface whenever the REE is capable of doing so.

In essence, this means that an attacker cannot disrupt or block TEE-based communication without breaking connectivity for the REE as well. It is important to note, however, that the attacker may still be able to disrupt the interface for both the TEE and the REE entirely. However, this also prevents the attacker from exploiting the device for malicious network traffic [4], [16].

5. Design and Implementation

Having high-end IoT devices, networking infrastructure, industrial control systems, and automotive systems in mind, we design a system with an Ethernet interface capable of sending and receiving frames from within the TEE and show that it complies with CNA. We base our system on an ARMv8-A device with TrustZone support.

Overview. Fig. 1 gives an overview of our system architecture. Following related work [32], [37], we propose a split-driver model and keep the majority of the NIC

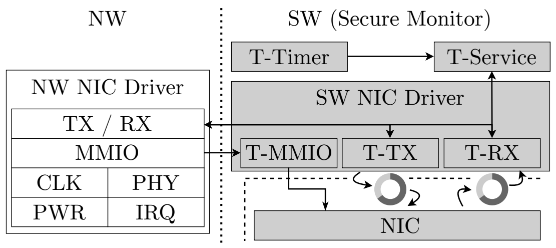

Figure 1. Overview of the proposed architecture. Arrows indicate data flow. The dotted line shows the isolation between NW and SW. The dashed lines indicate the interface between software and hardware. Components with a grey background are in the TCB.

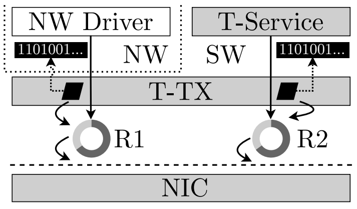

Figure 2. TX interface. Solid arrows indicate data flow. Dashed arrows are pointers. Black parallelograms depict descriptors. Black rectangles are buffers for frames. The dotted line shows the isolation between NW and SW. The dashed lines indicate the interface between software and hardware. Components with a grey background are in the TCB.

driver in the NW to keep the TCB small. For example, clock handling (CLK), power management (PWR), PHY management [13], and IRQ handling remain in the NW. In contrast, we split the MMIO register handling, the TX processing, and the RX processing into one module in the NW and one in the SW (prefixed with *T-* for *trusted*). Together, `T-MMIO`, `T-TX`, and `T-RX` provide an Ethernet interface within the TEE, which we multiplex between the worlds so that the NW cannot selectively block SW traffic. After the NIC has been initialized via setting MMIO registers, the majority of the communication takes place via the rings (Sec. 2). However, we cannot keep the rings in the NW. This is because a compromised NW could overwrite SW descriptors before they are processed, breaking CNA guarantees. Thus, `T-MMIO` intercepts the access to the NIC configuration to ensure that the rings are in SW memory. We describe the module in Sec. 5.1 in more detail. The `T-TX` module offers an interface to SW and NW that enables the transmission of an Ethernet frame. The module inserts a buffer descriptor to the right ring at the right position while ensuring that CNA holds. We deal with TX processing in Sec. 5.2. Similarly, `T-RX` provides receiving capabilities for NW and SW, while ensuring CNA (Sec. 5.3). `T-Service` is an *exemplary* trusted service that relies on this interface. For the sake of simplicity, we directly include `T-Service` in the secure monitor in our Proof of Concept (PoC). However, we argue that, without loss of generality, the interface can be exported to the rest of the SW, i.e., the SW OS and its applications, as well. In the current state, `T-Service` contains logic for benchmarking purposes (Sec. 6).

Available Trusted Service. CNA requires computational availability for the SW. Otherwise, the NW can starve the TEE (Sec. 1). Following related work [37], [39], we use a trusted timer (`T-Timer`) to trap the execution flow to the SW regularly. Once the timer IRQ fires, we execute our trusted service. Subsequently, the secure monitor passes the control flow back to the NW. Since the NW is not able to interfere with the timer or the IRQ, the trusted service is computationally available.

5.1. Controlling the NIC Configuration

As a first step towards CNA, we move the descriptor rings and the contained buffer descriptors from NW memory to SW memory. To do so, we first alter the secure monitor so that the NIC is assigned to the SW during the boot process, thereby permitting access to the NIC MMIO registers exclusively to the SW software. Note that the assignment happens before the NW starts.

After that, we alter the NIC driver in the NW so that every access (e.g., `readl` or `writel`) to a NIC MMIO register issues a SMC, resulting in a context switch to `T-MMIO`. Subsequently, `T-MMIO` can monitor and modify the configuration changes before it writes the values to the NIC registers. In particular, the `T-MMIO` module writes pointers to the SW descriptor rings to the configuration registers and blocks any attempt of the NW to change to location of the descriptor rings in the memory. Since we control the NIC MMIO registers, we can enforce that the NIC uses the rings in the SW. Since they are stored in SW memory, the NW no longer has direct access to read from or write to the descriptor rings.

To preserve connectivity for the NW, we alter the NW NIC driver so that an SMC to `T-TX` (Sec. 5.2) resp. `T-RX` (Sec. 5.3) is issued whenever the NW driver intends to write a descriptor or read a descriptor from a ring.

5.2. TX Interface

Typically, NICs have multiple TX rings (Sec. 2), resulting in multiple strategies on how to fill the rings. In the single queue strategy, the OS uses one TX ring. However, for high throughputs (i.e., $\geq$10 Gbps), a single queue can lead to resource contention [33]. To overcome contention issues, multi-queue strategies are used, which assign TX rings to CPU cores. As a result, the cores do not need to synchronize access to a common TX ring.

Descriptor Ring for the SW. Instead of another CPU core, we propose to assign one TX ring to the SW (Fig. 2). A dedicated ring for the SW saves us from synchronizing the current position in the ring between NW and SW, which would entail additional complexity. Via calls to the `T-TX` module, the trusted service can prepare a buffer descriptor (black parallelogram) that points to a network frame in the SW (binary digits). Once the `T-TX` module inserts the descriptor into the TX descriptor ring dedicated to the SW (R2, Fig. 2), the frame is sent. In contrast, the network stack of the NW OS submits network frames to the NW NIC driver. Subsequently, the NIC driver assembles a buffer descriptor pointing to the network frame in NW memory. We modify the NIC driver in a way that the driver issues an SMC when the descriptor is ready for transmission. The `T-TX` module inserts the descriptor into one of the rings dedicated to the NW (R1, Fig. 2). Subsequently, the network frame is sent.

CNA for TX. To fulfill CNA guarantees for TX, we need to ensure that the NIC consistently transmits frames from the SW ring (R2, Fig. 2) whenever the NIC transmits frames from the rings assigned to the NW (R1, Fig. 2). Essentially, this requires preventing NIC configurations that exclusively disable or impede the SW TX ring while maintaining the functionality of the NW ring(s).

With the help of the `T-MMIO` module, we block any attempt from the NW to (1) configure the NIC to use other TX rings than those that we set up in the SW, (2) disable the SW TX ring, (3) change the structure or size of descriptor rings, (4) decrease the maximum TX size to unreasonable values, and (5) change the interpretation of the frames in RAM (e.g., padding bytes). Finally, (6) detecting a NIC reset is crucial, which typically resets the positions within the descriptor rings where the NIC anticipates the next buffer descriptor. Consequently, failing to detect resets leads to the insertion of descriptors at the wrong positions in the TX ring. The NIC waits for the descriptor at another position to be ready, potentially breaking network connectivity for the SW only. Since a NIC reset is typically triggered via writing a MMIO register, we can use `T-MMIO` to reliably detect NIC resets and restore our bookkeeping of the current positions in the rings to the initial values.

In our practical observations, the NIC driver sets the MMIO registers responsible for the six requirements from the previous paragraph to a static value during initialization. We extract the expected values from the code and design the `T-MMIO` module so that other values for the MMIO registers are rejected. Moreover, verifying selected registers does not come with great additions to the TCB, leaving the driver's complexity in the NW (Sec. 7).

With `T-MMIO` in place, we can not only guarantee that the NIC enables and uses the TX ring dedicated for the SW, but also enforce a certain structure of the descriptors in the ring. Thus, the `T-Service` can prepare network frames and issue requests to `T-TX` to insert descriptors (with the expected structure) to the SW TX ring pointing to the frames. Since the NW cannot selectively disable the SW TX ring, the frames are sent as long as the NIC and the network allow sending frames at all. Thus, our TX design complies with CNA.

Mitigating Malicious Pointers. Particular caution is required when pointers are exchanged across isolation boundaries, e.g., when the privileged SW dereferences pointers passed from the less-privileged NW. This is because the SW lacks semantic information about the passed pointers. Without proper validation that the pointer references NW memory, the NW can trick the SW to dereference pointers to the SW, often resulting in privilege escalation attacks [9]. Under normal circumstances, the descriptors in the NW TX point to network frames in NW memory (Fig. 2). However, maliciously injected pointers could allow the attacker to manipulate the NIC into reading (and potentially even sending) SW memory.

Whenever the NW requests to insert a descriptor in the ring (R1, Fig. 2), we first copy the descriptor to scratch memory in the SW. We then validate that the pointer in the descriptor references NW memory and also check the frame's length (also stored in the descriptor). If the descriptor is legitimate, we set the ready flag and insert the descriptor into the ring, giving the NIC permission to process the descriptor and the frame. The attacker cannot exchange the pointer after the validation since the descriptor is in SW memory, thus preventing Time-of-check Time-of-use (TOCTOU) attacks.

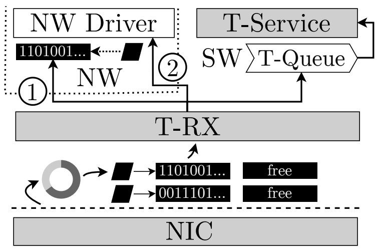

Figure 3. RX interface. Solid arrows indicate data flow. Dashed arrows are pointers. Black parallelograms depict descriptors. Black rectangles are buffers for frames. The dotted line shows the isolation between NW and SW. The dashed lines indicate the interface between software and hardware. Components with a grey background are in the TCB.

TX IRQs. For general-purpose hardware, the NW configures the NIC to issue an IRQ once a frame is transmitted (Sec. 2). However, since our trusted service now also transmits frames, we would generate IRQs, resulting in additional (and unnecessary) calls to the IRQ handler in the NW OS and, thus, jitter. As an alternative, one could route the NIC IRQs to the SW. In this case, however, any of the IRQs originating from NW transmissions are irrelevant to the SW. One would need to inject an IRQ into the NW [20], resulting in overhead as well. Moreover, handling and forwarding IRQs tends to be complex and error-prone, and we want to keep a small TCB.

Thus, we choose to not change the IRQ behavior. If the NW transmits a frame, an IRQ is handled in the NW as usual. For the SW traffic, we propose the following strategy. NIC IRQs can add a significant load on the system. Therefore, hardware features that can precisely control IRQ generation are widely available. For example, most NICs contain an IRQ flag in the descriptor. Only if the flag is set, the NIC issues IRQs if the descriptor has been processed. We propose to clear the IRQ flag in the descriptors inserted into the SW TX ring. Thus, no IRQs are issued for frames sent from the SW, effectively leading to a polling approach for the SW [37].

Overall, we argue that assigning one TX ring for SW traffic is justifiable considering the additional security guarantees and depending on the specific use case. In our evaluation (Sec. 6), we show that the practical consequences on the NW traffic are small.

5.3. RX Interface

Some NICs support to route frames to a RX ring based on packet headers [27]. Thus, it is possible to apply a similar strategy. We could configure the NIC so that SW frames arrive in a dedicated ring. However, we have decided against this strategy for the following reasons. First, we observe that NICs do not necessarily have this capability. Second, this approach also hinders flexibility since the NIC's routing capabilities dictate the number of data flows we can receive in the SW ring.

We propose a more flexible approach (Fig. 3). Upon initialization, `T-RX` prepares the RX rings so that the descriptors point to buffers in SW memory. When the NIC receives a frame, the NIC copies it to the next free buffer and issues an IRQ. Subsequently, the (modified) driver in the NW OS tries to read the next buffer descriptor in the RX ring and issues an SMC to `T-RX`, passing a pointer to an empty NW buffer and its length as a parameter.

The `T-RX` module decides on the destination of the received frame. In the current PoC, we expect the device owner to allocate a fixed port for a trusted service. We parse the IP and UDP header and compare the port number with the trusted service. In a more feature-rich system, trusted services should be able to listen on protocols and ports, similar to the socket interface in the NW. We believe that such mechanisms can be integrated with a sandboxed TCP/IP stack (Sec. 7). In the case that the frame is intended for the trusted service, the `T-RX` module copies the frame to an input queue (`T-Queue`, Fig. 3), clearing the receive buffer for new arrivals. We rely on an array of frame buffers for the queue. Compared to a list (with a memory allocator), this only comes with small additions to the TCB. We set the size to $1MiB$. A fitting size can be determined empirically.

During the next timer tick, the service processes the data from the queue. From the perspective of the NW, the buffer still contains zeroes after the SMC. For the NW driver, it appears as if it has received a packet consisting of zeros. Since this is not a valid packet, it is dropped.

We route a frame to the NW if it is not intended for the SW. We first validate that the pointer to the NW buffer references NW memory with a given length. The NIC stores the length of the received frame in the descriptor. Subsequently, we answer the NW driver's request to the next buffer descriptor in the RX ring with a descriptor that points to the NW buffer (2, Fig. 3) instead of the SW buffer. We do not interfere with RX IRQs (Sec. 5.2).

CNA for RX. Our RX interface complies with CNA if we can ensure that the NW can only process received frames if the SW can do so as well (Sec. 4). Similar to TX (Sec. 5.2), the `T-MMIO` module blocks any attempt from the NW to (1) configure the NIC to use other RX rings, (2) change the structure or size of RX rings, (3) unreasonably decrease the maximum frame size, and (4) change the interpretation of the frames in RAM. `T-MMIO` can reliably detect a NIC reset to refresh the positions at which we expect the descriptor of the next received frame.

Every frame that the NIC receives is directly written to SW memory to which the NW has no access. Due to the `T-MMIO` module, an attacker cannot force the NIC to use malicious rings in the NW. Thus, the only option to access the RX rings and receive data in the NW is via the `T-RX` module. However, if the `T-RX` module is called via an SMC, received SW frames are processed, inserted into `T-Queue`, and forwarded to the trusted service. Therefore, the RX interface complies with CNA if SW frames can be inserted into `T-Queue`.

We pay special attention to the case that `T-Queue` is full. When the NIC receives a SW frame and the queue is full, the frame is discarded as common in networking. Notably, we also drop a received frame for the NW if the queue is full, even though the frame would never have been inserted into `T-Queue`. This way we ensure that the RX interface complies with CNA during congestion.

We note that an attacker might choose to refrain from issuing a SMC to handle a RX IRQ. Consequently, `T-RX` is not invoked. Strikingly, this does not break CNA, since the NW cannot receive any frames. Alternatively, we could also use a polling approach similar to TX and use a queue to pass traffic to the NW.

5.4. Optimizations

Dynamic Timer Frequency. We do not rely on constant timer ticks for `T-Service` but instead utilize dynamic timer ticks. With each invocation of `T-Service`, we measure the fill levels of the descriptor rings. If the levels exceed a configurable threshold, we assume a high volume of traffic and increase the timer frequency (up to a configurable maximum) to process the traffic more quickly. Conversely, if the fill levels fall, we reduce the timer frequency again (down to a configurable minimum). By tweaking these settings, one can flexibly optimize our system for maximum performance or minimal jitter.

Batch Processing Descriptors. Inspired by existing literature [32], we implement the `T-TX` and the `T-RX` module to work with batches of descriptors, reducing the number of context switches between the worlds. With one transmit request of the NW, for example, we process every descriptor that is ready for transmission at that time.

6. Evaluation

6.1. Proof of Concept

We implement a PoC on a Nitrogen8M board with an NXP i.MX8M ARMv8-A CPU and 2GB of RAM. The board has a Freescale Ethernet NIC (Atheros AR8035 PHY), which supports speeds up to 1000 Mbit/s. The Central Security Unit (CSU) allows assigning peripherals to a world. We use the i.MX fork of Trusted Firmware-A (TFA), v2.2, as a secure monitor (Sec. 5) and assign the NIC to the SW during the boot process, before the NW OS is started. In the NW, we use Linux (v5.10.63).

We implement our modules `T-MMIO`, `T-TX`, `T-RX`, `T-Timer` (Fig. 1), and `T-Queue` (Fig. 3) as C modules linked to TFA. We register entry points for SMCs in `T-MMIO`, `T-TX`, and `T-RX` with TFA, which allows the NW to call them. In our PoC, we also link `T-Service` directly to TFA. Without loss of generality, the CNA-compliant Ethernet interface can be exported to the rest of the SW, i.e., the SW OS and its applications, as well. We enable the timer in TFA and call `T-Service` in the timer handler. We replace the `memcpy` in TFA with an optimized version from ARM [5].

6.2. Performance

We evaluate our system in a test network. As a router, we use a lightly loaded AVM FritzBox! 7390. We set up an IPv4 network where the devices are in the same subnet and are assigned static addresses. We connect our development board and a ThinkPad T14s laptop, subsequently referred to as the *host*, via Ethernet to the router. The host has a Core i7-10610U CPU, 32GB DDR4 RAM, and an Intel I219-V NIC. Our baseline is a system without any modifications to the network stack or the SW.

We implement a packet generator that saturates the link with MTU-sized UDP packets. We also implement a receiver, which calculates the throughput over the last $10s$.

TABLE 1. SINGLE FLOW PERFORMANCE.

Experiment	TX		RX	
	Baseline	Ours	Baseline	Ours
T_{NW} [Mbit/s]	956.41	956.32	947.86	949.05
T_{SW} [Mbit/s]	-	937.24	-	948.78
L_{NW} [ms]	1.65	1.61	1.60	1.59
L_{SW} [ms]	-	10.00	-	9.87

TABLE 2. MULTI FLOW PERFORMANCE.

Experiment	NW	SW	Sum
S_{TX} [Mbit/s]	20.95	935.46	956.51
S_{RX} [Mbit/s]	476.72	480.39	957.11

For the generator and receiver, we implement a variant for Linux and for our trusted service. For each experiment, we measure 20 iterations and average the results. The standard deviation is insignificant in every experiment.

Additionally, we measure the latency in both directions and for both worlds by echoing a single UDP packet while measuring the RTT. Again, we measure 20 iterations and average the results, with insignificant standard deviations.

For our baseline, we measure the NW throughput T_{NW} when the device runs the generator and the host the receiver (TX, Tab. 1) or vice versa (RX, Tab. 1). We measure the NW latency L_{NW} when the device sends the test packet (TX, Tab. 1) or echos it back (RX, Tab. 1). We repeat T_{NW} and L_{NW} with our system (*Ours*, Tab. 1) and also measure the throughput and latency of our trusted service (T_{SW} and L_{SW}).

As Tab. 1 shows, our system does barely impact the throughput or latency of NW traffic. Interestingly, we even measure a slightly higher NW throughput during RX. We suspect that the aggressive and uninterruptable batch processing of incoming frames in the SW is the reason.

We reach 98% of the throughput in our trusted service when transmitting frames and 99.97% when receiving data. Compared to the NW, our trusted service has a higher network latency. We insert the frame into a queue (Sec. 5.3) and process it only at the next timer tick. The value of around $10ms$ matches the timer frequency in our latency experiment and, thus, our expectations. We note that for use cases in the realm of high-end IoT devices, networking infrastructure, industrial control systems, and automotive systems (Sec. 5), this latency is in the magnitude of pinging a remote server. If lower latencies are required, one could increase the execution frequency of the trusted service (Sec. 5.4) at the expense of higher jitter for the NW OS, or switch to an IRQ-based approach in the future (Sec. 7). Despite the higher latency, the system achieves near-native throughput. We deem the batch processing of buffer descriptors responsible (Sec. 5.4).

To show that the NW cannot monopolize the NIC by excessive traffic, we also measure multiple simultaneous flows (Tab. 2). S_{TX} shows the throughput when we send frames in the SW and the NW concurrently. Through aggressive sending and polling, we prioritize SW traffic. Consequently, the NW can only send data at $21Mbit/s$ if the SW continuously transmits data (without any break). We scale the frequency between $20Hz$ and $170Hz$. During the throughput benchmarks, the frequency is gradually increased to $170Hz$, which we empirically determine as the lowest frequency at which the SW saturates the link. By tweaking the frequency settings, the prioritization of SW traffic can be adjusted to the use case.

In S_{RX}, we add a second host (a second Nitrogen8M board) to the network. Subsequently, one host transmits data to the SW of our evaluation board, while the other transmits data to the NW. Both send at full line rate. We observe that the throughput per world is roughly cut in half and presume that the router distributes the traffic in a round-robin fashion. S_{RX} shows that if SW frames reach the device, they are processed alongside the NW frames, even if the link is saturated. This conforms to CNA.

6.3. Trusted Computing Base

We use `cloc` (version 1.82) to count the LoC of our core modules `T-Timer`, `T-MMIO`, `T-TX`, `T-RX` (Fig. 1), and `T-Queue` (Fig. 3). Our modules comprise only 992 LoC in total. Together, these provide a CNA-capable Ethernet interface to the SW. `T-Service` is an exemplary trusted service and contains logic for performance benchmarking purposes, which is not required for functionality. To implement a trusted service, one would either replace this code with the actual application logic or export the CNA-compliant Ethernet interface to the SW OS and link the trusted service against it. Of course, depending on the size of the actual trusted service, the TCB increases. Therefore, we do not include `T-Service` in our calculations. We note that TFA, on which we base our system, comprises 12.5 kLoC. Therefore, we increase the TCB by less than 7.93%. We also note that the Ethernet driver in the NW consists of 4.3 kLoC, showing that our split-driver approach effectively reduces the additions to the TCB.

7. Discussion and Future Work

Portability. Our system requires ARM TrustZone and a DMA-based NIC with multiple TX rings and fine-granular IRQ control. Most NICs use DMA [37]. To the best of our knowledge, most modern NICs support multiple TX queues and fine-granular IRQ control, suggesting a wide applicability. Porting our system to other platforms is left for future work. Moreover, we are currently focusing on systems with a single NIC. For systems with multiple NICs, CNA can be applied individually to each of them. We believe that synchronization between the NICs allows achieving system-wide CNA for multi-NIC systems. The integration is future work.

Denial of Service. An attacker can either flood the device with NW traffic or have determined which traffic is routed to the SW and attempt to flood the SW. If the traffic is passed to the NW, the network stack consumes CPU and memory. In the case of SW traffic, the data is processed during the next timer tick, which also consumes resources. We follow existing literature [17], [40] and argue that denial-of-service attacks can (only) be countered at the infrastructure level, which is out of our scope (Sec. 3). We still discuss one particular attack vector. The attacker could attempt to flood `T-Queue` (Sec. 5.3) to cause legitimate SW frames to be dropped (due to a full queue) while allowing malicious NW frames to continue being processed. To rule out this attack, we only forward frames to the NW if the `T-Queue` is not already filled, ensuring CNA for the RX interface.

Timed Attacks. By flooding the device with traffic in time patterns, an adversary can enforce windows during which they can communicate with the device (e.g., to transmit new commands), and block any communication otherwise. However, we argue that (1) flooding attacks may be mitigated at the infrastructure level and (2) SW traffic is still processed within the time windows, conforming to CNA. Nevertheless, timed attacks should be kept in mind when implementing trusted services. Achieving unconditional network availability (for trusted services) is out of scope if not impossible (Sec. 3).

IRQ-Based SW Interface. Generally speaking, a polling-based approach for handling the SW traffic wastes CPU cycles and slows the scheduling of NW processes on the same core. In this work, we deliberately refrain from an IRQ-based approach. This is because handling and distributing IRQs between the NW and the SW would add a lot more complexity, is error-prone, and increases the TCB. To mitigate the negative effects, we propose a dynamic polling frequency (Sec. 5.4). Nevertheless, options for IRQ-based designs are interesting for the future.

Complex Transmission Protocols. Currently, our system processes Ethernet frames in the SW. More complex protocols such as TCP/IP are not yet supported. However, we note that we have successfully solved the challenging task of secure CPU-to-NIC communication and can effectively multiplex a single NIC between the worlds while adhering to CNA. Building upon Ethernet send and receive primitives, further protocols can be implemented in an isolated address space [18], [42], [44] or an enclave [28], [36], shielded from any peripheral and other software. The integration is an interesting direction for future work.

8. Related Work

TZNIC is most closely related to our system [37]. Wan et al. follow similar goals, suggesting multiplexing the NIC between NW and SW. They adopt an approach that requires no modifications to the NW, while we propose small changes. However, the most decisive difference is that TZNIC only supports receiving frames in the SW, whereas we support both transmitting and receiving frames. With TZNIC, a sender might need to transmit a frame multiple times until the SW can get hold of it. This is due to the NW dealing with the packets before the SW can process them. Our system is immune to this limitation, since the NIC directly stores the received frames in the SW, to which the NW has no access.

Focusing on consumer routers, Schwarz splits the NIC driver and moves some components (e.g., routing and firewalling) to the SW to protect their integrity [32]. While Schwarz does not focus on any form of network availability, we believe that CNA could be integrated.

Software-based solutions like VirtIO [30], which can provide a virtualized NIC to a Virtual Machine (VM), are widely available. Typically, the hypervisor includes a fully-featured driver for a physical NIC and multiplexes the traffic. Most systems supporting TrustZone inherently support assigning peripherals to the SW. However, assigning a device to a TEE becomes challenging if the device is connected to a complex bus system (e.g., PCIe). Hardware-based solutions like Single Root I/O Virtualization (SR-IOV) address this and allow dividing a single physical NIC into multiple Virtual Functions (VFs), enabling direct communication from a VM to the NIC. Further, the TEE Device Interface Secure Protocol (TDISP) allows establishing a trust relationship between an isolated execution environment and a peripheral, while encrypting the data flow between them [25]. Besides hypervisors, other high-privileged execution contexts have been used for networking as well. Wang et al. [38] and Zhang et al. [43] propose to include NIC drivers in the System Management Mode (SMM) on x86/AMD64 and Liu et al. [23] implement a complete NIC driver in the TrustZone. All these approaches introduce a complex NIC to the TCB. In contrast, we follow an existing line of argumentation [20], [32], [37] and rely on a split-driver model to keep the additions to the TCB small.

As an alternative to splitting drivers, several approaches propose record-and-replay as a strategy to generate TEE-based drivers [15], [41]. They replicate device interactions using previously recorded behavior templates. However, NIC drivers are complex and heavily rely on DMA and IRQs, posing a challenge for these approaches. While Guo et al. state NIC drivers are out of scope [15], the authors of LDR believe that their system might, with additional work, be applicable to NIC drivers [41].

9. Conclusion

We observe that trusted services often require communication with a trusted party. However, typical systems rely on a network agent in the REE to forward traffic, giving up any form of network availability. To prevent an NW attacker from monopolizing the network channel, we introduce CNA as a novel concept for TEE-based network I/O. Based on a split-driver approach, we multiplex a complex Ethernet NIC between TEE and REE while guaranteeing that the TEE can process network I/O whenever the REE can. CNA has applications in remote device management, system monitoring, and intrusion detection, where obstructing TEE connectivity can lead to adverse system states. We show that our system achieves near-native throughput while keeping the TCB small.

Major upcoming architectures for confidential computing (e.g., AMD SEV-SNP [2], Intel TDX [11], and ARM CCA [21]) primarily focus on confidentiality and integrity, whereas availability is less prominent. This leaves us with similar conditions for network I/O. Therefore, we believe that both the concept of CNA and our implementation strategy are relevant for describing and building secure systems in the future.

Acknowledgements

We thank the reviewers and our shepherd for the valuable feedback. This research was partly supported by the German Federal Ministry of Education and Research (BMBF) as part of the CELTIC-NEXT project AI-NET-ANTILLAS ("Automated Network Telecom Infrastructure with inteLLigent Autonomous Systems", Förderkennzeichen "16KIS1314").

References

[1] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security, CCS '16*, pages 743–754. ACM, 2016.

[2] Advanced Micro Devices Inc. AMD Secure Encrypted Virtualization (SEV). https://www.amd.com/de/developer/sev.html, 2020. Accessed 2024-03-17.

[3] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves. In *Proceedings of the 27rd ACM Conference on Computer and Communications Security, CCS '21*, pages 1357–1372. ACM, 2021.

[4] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium, USENIX Sec '17*, pages 1093–1110. USENIX Association, 2017.

[5] ARM Limited. ARM Optimized Routines. https://github.com/ARM-software/optimized-routines, 2017. Accessed 2024-03-08.

[6] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference, DAC '15*, pages 34:1–34:6. ACM, 2015.

[7] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. TEEzz: Fuzzing Trusted Applications on COTS Android Devices. In *Proceedings of the 44th IEEE Symposium on Security and Privacy, S&P '23*, pages 1204–1219. IEEE, 2023.

[8] Marcel Busch, Ralph Schlenk, and Hans Heckel. TEEMo: trusted peripheral monitoring for optical networks and beyond. In *Proceedings of the 4th Workshop on System Software for Trusted Execution, SysTEX '19*, pages 7:1–7:6. ACM, 2019.

[9] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy, S&P '20*, pages 1416–1432. IEEE, 2020.

[10] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS '18*. The Internet Society, 2018.

[11] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX Demystified: A Top-Down Approach. *CoRR*, abs/2303.15540, 2023.

[12] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium, USENIX Security '16*, pages 857–874. USENIX Association, 2016.

[13] Linux Kernel Developers. PHY Abstraction Layer. https://www.kernel.org/doc/Documentation/networking/phy.txt, 2008. Accessed 2024-04-20.

[14] GlobalPlatform Inc. TEE Sockets API Specification 1.0. https://globalplatform.org/wp-content/uploads/2017/01/GPD_TEE-Sockets-API-_v1.0.pdf, 2015. Accessed 2024-03-11.

[15] Liwei Guo and Felix Xiaozhu Lin. Minimum viable device drivers for ARM trustzone. In *Proceedings of the 17th European Conference on Computer Systems, EuroSys '22*, pages 300–316. ACM, 2022.

[16] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS '19*. The Internet Society, 2019.

[17] Manuel Huber, Stefan Hristozov, Simon Ott, Vasil Sarafov, and Marcus Peinado. The Lazarus Effect: Healing Compromised Devices in the Internet of Small Things. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, ASIA CCS '20*, pages 6–19. ACM, 2020.

[18] Eunyoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI '14*, pages 489–502. USENIX Association, 2014.

[19] Benedikt Jung, Christian Eichler, Jonas Röckl, Ralph Schlenk, Timo Hönig, and Tilo Müller. Trusted Monitor: TEE-Based System Monitoring. In *Proceedings of the 12th Brazilian Symposium on Computing Systems Engineering, SBESC '22*, pages 1–8. IEEE, 2022.

[20] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. SeCloak: ARM Trustzone-based Mobile Peripheral Control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '18*, pages 1–13. ACM, 2018.

[21] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI '22*, pages 465–484. USENIX Association, 2022.

[22] Linaro Limited. OP-TEE Documentation. https://optee.readthedocs.io/en/latest/index.html, 2023. Accessed 2024-03-13.

[23] Dongtao Liu and Landon P. Cox. VeriUI: attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications, HotMobile '14*, pages 7:1–7:6. ACM, 2014.

[24] Ramya Jayaram Masti, Claudio Marforio, Aanjhan Ranganathan, Aurélien Francillon, and Srdjan Capkun. Enabling trusted scheduling in embedded systems. In *Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12*, pages 61–70. ACM, 2012.

[25] PCI-SIG. TEE Device Interface Security Protocol (TDISP). https://pcisig.com/tee-device-interface-security-protocol-tdisp, 2022. Accessed 2024-04-24.

[26] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems. *IEEE Comput. Archit. Lett.*, 16(2):158–161, 2017.

[27] Solal Pirelli and George Candea. A Simpler and Faster NIC Driver Model for Network Functions. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI '20*, pages 225–241. USENIX Association, 2020.

[28] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter R. Pietzuch. SGX-LKL: Securing the Host OS Interface for Trusted Execution. *CoRR*, abs/1908.11143, 2019.

[29] Jonas Röckl, Mykolai Protsenko, Monika Huber, Tilo Müller, and Felix C. Freiling. Advanced System Resiliency Based on Virtualization Techniques for IoT Devices. In *Proceedings of the 37th Annual Computer Security Applications Conference, ACSAC '21*, pages 455–467. ACM, 2021.

[30] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[31] Samsung Electronics Co.Ltd. Whitepaper: Samsung Knox Security Solution. https://images.samsung.com/is/content/samsung/p5/global/business/mobile/SamsungKnoxSecuritySolution.pdf, 2017. Accessed 2024-02-13.

[32] Fabian Schwarz. TrustedGateway: TEE-Assisted Routing and Firewall Enforcement Using ARM TrustZone. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses, RAID '22*, pages 56–71. ACM, 2022.

[33] Brent E. Stephens, Arjun Singhvi, Aditya Akella, and Michael M. Swift. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC '17*, pages 431–444. USENIX Association, 2017.

[34] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. OAT: Attesting Operation Integrity of Embedded Devices. In *Proceedings of the 41st IEEE Symposium on Security and Privacy, S&P '20*, pages 1433–1449. IEEE, 2020.

[35] JSOF Tech. Ripple20 - CVE-2020-11901. https://www.jsof-tech.com/wp-content/uploads/2020/08/Ripple20_CVE-2020-11901-August20.pdf, 2020. Accessed 2024-04-25.

[36] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter R. Pietzuch. rkt-io: a direct I/O stack for shielded execution. In *Proceedings of the Sixteenth European Conference on Computer Systems, EuroSys '21*, pages 490–506. ACM, 2021.

[37] Shengye Wan, Kun Sun, Ning Zhang, and Yue Li. Remotely controlling TrustZone applications?: a study on securely and resiliently receiving remote commands. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec '21*, pages 204–215. ACM, 2021.

[38] Jiang Wang, Fengwei Zhang, Kun Sun, and Angelos Stavrou. Firmware-assisted Memory Acquisition and Analysis tools for Digital Forensics. In *Proceedings of the Sixth International Workshop on Systematic Approaches to Digital Forensic Engineering, SADFE '11*, pages 1–5. IEEE Computer Society, 2011.

[39] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy, S&P '22*, pages 352–369. IEEE, 2022.

[40] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. Dominance as a New Trusted Computing Primitive for the Internet of Things. In *Proceedings of the 40th IEEE Symposium on Security and Privacy, S&P '19*, pages 1415–1430. IEEE, 2019.

[41] Huaiyu Yan, Zhen Ling, Haobo Li, Lan Luo, Xinhui Shao, Kai Dong, Ping Jiang, Ming Yang, Junzhou Luo, and Xinwen Fu. LDR: Secure and Efficient Linux Driver Runtime for Embedded TEE Systems. In *Proceedings of the 2024 Annual Network and Distributed System Security Symposium, NDSS '24*. The Internet Society, 2024.

[42] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The True Cost of Containing: A gVisor Case Study. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud '19*. USENIX Association, 2019.

[43] Fengwei Zhang, Jiang Wang, Kun Sun, and Angelos Stavrou. HyperCheck: A Hardware-Assisted Integrity Monitor. *IEEE Trans. Dependable Secur. Comput.*, 11(4):332–344, 2014.

[44] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu. Deploying User-space TCP at Cloud Scale with LUNA. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC '23*, pages 673–687. USENIX Association, 2023.