Hardening VirtIO Drivers for Confidential VMs with Rust-Based Typed Interfaces

Lianjie Wang[1,a)] Takahiro Shinagawa[1,b)]

Abstract: Confidential virtual machines (CVMs) isolate sensitive workloads from untrusted hosts but still rely on them for I/O through VirtIO devices. Most existing VirtIO drivers assume trusted hosts and thus trust device-supplied fields in shared memory that hosts can rewrite at any time, as recent Linux driver vulnerabilities demonstrate. Existing hardening efforts add validation checks to individual drivers, but these checks can be incomplete or bypassed by time-of-check to time-of-use (TOCTOU) attacks. We propose a framework that hardens VirtIO drivers by construction using Rust's type system. The framework encapsulates VirtIO primitives behind typed interfaces that enforce buffer ownership and exposes host-writable memory only through a forward-only cursor that fetches each byte at most once. A domain-specific language (DSL) based on Rust procedural macros generates parsers from struct definitions that validate fields in private memory and reject inputs that violate declared constraints, preventing buffer overflows and double-fetch vulnerabilities. We implemented the framework using Rust for Linux, ported the virtio-rng and virtio-vsock drivers to it, and modified the existing virtio-net and virtio-blk drivers to use DSL-generated validators through a foreign function interface (FFI). We conducted a large language model (LLM)-assisted audit of the mainline VirtIO drivers and identified 15 defects in 12 drivers, 11 of which had no complete fix posted before our audit. The framework's mechanisms could prevent 12 of these defects, three of which we experimentally validated using the implemented drivers. The validators rejected malformed device input injected by a modified QEMU, including a replay of CVE-2025-39718. A targeted TOCTOU experiment showed that the Rust virtio-vsock driver avoided inconsistent reads observed in the C drivers. During 24 hours of fuzzing with mutated packets, the Rust driver triggered no crashes. Its median receive throughput was 2.4–6.5% lower than that of the C driver under the original refill policy, but differed by at most 1.6% when both drivers used an optimized refill policy. We also evaluated our typed abstractions using a Rust model of the kernel interfaces under Miri, which identified five concurrency and lifecycle issues that we subsequently fixed.

Keywords: Confidential VMs, VirtIO, Rust for Linux, input validation, device drivers

1. Introduction

Confidential virtual machines (CVMs) isolate sensitive workloads from untrusted cloud providers by encrypting guest memory and CPU state, as in Intel Trust Domain Extensions (TDX) and AMD Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP). However, virtual I/O devices are still provided by the untrusted host, typically through VirtIO [1], where the host emulates network, block, and socket devices and writes their data into shared, unencrypted memory. Therefore, the CVM threat model treats all device-supplied values as potentially malicious [2]. Most existing VirtIO drivers, however, were written assuming a trusted host, and recent vulnerabilities have exposed the risks of this assumption. For example, the Linux virtio-vsock driver failed to validate the payload length in received packet headers, allowing an out-of-bounds write [3]. Similarly, a length check in virtio-net was fixed in Linux 6.18 [4] but required another fix in Linux 7.3 [5], because the first fix still accepted lengths exceeding the buffer size, leading to an out-of-bounds write. VIA, a fuzzer for the device interfaces of protected VMs, analyzed 22 Linux drivers and found 50 such bugs [6].

These bugs reflect a reliance on manual validation of device input. For example, Figure 1 shows that the virtio-vsock driver checks the length returned by `virtqueue_get_buf()` but passes the separate, unchecked payload length from the packet header to `skb_put()`. In C drivers, the language neither enforces validation of device-written fields before use nor prevents *double fetches*, in which drivers reread fields from shared memory after validation. The host can exploit double fetches in time-of-check to time-of-use (TOCTOU) attacks by changing fields between validation and use. Wang et al. found 90 double-fetch situations in the Linux kernel, including 57 in drivers [7]. We also conducted an audit of VirtIO drivers in Linux 7.3 with assistance from a large language model (LLM) and identified 15 defects in 12 drivers, demonstrating that vulnerabilities in handling untrusted device input persist even in recent kernels.

Existing approaches address these problems only par-

[1] The University of Tokyo
[a)] lian@is.s.u-tokyo.ac.jp
[b)] shina@is.s.u-tokyo.ac.jp

ⓒ2026 Information Processing Society of Japan

```
/* virtio_transport_rx_work() */
skb = virtqueue_get_buf(vq, &len);
...
if (len < sizeof(struct virtio_vsock_hdr) ||
    len > virtio_vsock_skb_len(skb)) {
    kfree_skb(skb);
    continue;
}
virtio_vsock_skb_rx_put(skb);

static inline void
virtio_vsock_skb_rx_put(struct sk_buff *skb)
{
    u32 len;

    len = le32_to_cpu(virtio_vsock_hdr(skb)->len);
    if (len > 0)
        skb_put(skb, len);
}
```

Fig. 1 The receive path of the virtio-vsock driver before Linux 6.17, which checks the device-reported length but passes the unchecked payload length from the packet header to `skb_put()`.

tially. The upstream hardening effort for Intel TDX audits drivers, fuzzes the host interface, and adds missing checks to C code one by one [8], but missed fields and insufficient checks can leave vulnerabilities exploitable. Moreover, 12 of over 40 VirtIO hardening commits reverted or amended earlier fixes [9]. Rust for Linux [10] supports memory-safe driver development, but type safety alone does not enforce device-specific constraints, and a study of 240 Linux driver vulnerabilities found 45 that Rust would not prevent, including protocol violations and semantic errors [11]. EverParse3D generates memory-safe validators proven free of double fetches [12], but these guarantees alone do not prevent calling C code from rereading shared memory after validation or accessing buffers still owned by the device.

We propose a framework that uses Rust's type system to enforce input validation and harden VirtIO drivers by construction. As shown in Figure 2, the framework combines typed VirtIO abstractions with parsers generated by a domain-specific language (DSL). The typed VirtIO abstractions consist of two main components: typed queues (`Queue`) and a forward-only cursor (`VolatileReader`). Typed queues take ownership of buffers posted to devices and return ownership to drivers only after the devices report completion. The cursor provides access to host-writable shared memory without exposing it as an ordinary byte slice, fetching each byte at most once.

The DSL uses Rust procedural macros to generate parsers from struct definitions annotated with constraints. The generated parsers use `VolatileReader` to read fields into private memory, validate them, and reject inputs that violate the declared constraints. Rust drivers, such as virtio-rng and virtio-vsock, use these components through safe interfaces, while existing C drivers, such as virtio-net and virtio-blk, can invoke the generated parsers through foreign function interface (FFI) validation entries. The framework builds on the existing VirtIO core and other kernel implementations,

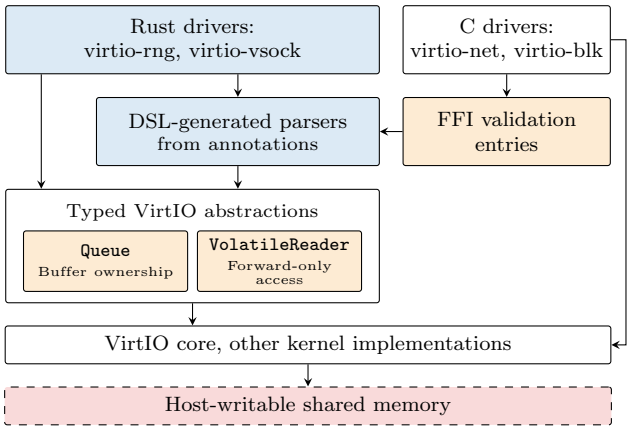

Fig. 2 The proposed framework architecture. Rust drivers use typed VirtIO abstractions and, where needed, DSL-generated parsers. The abstractions include `Queue` for buffer ownership and `VolatileReader` for forward-only access to host-writable memory. Existing C drivers use the generated parsers through FFI validation entries. The framework builds on the existing VirtIO core and other kernel implementations, which interact with host-writable shared memory. Orange boxes indicate Rust components containing `unsafe` code.

which interact with host-writable shared memory.

As a result, the generated parsers reject inputs that violate declared constraints, while the single-fetch guarantee prevents TOCTOU vulnerabilities caused by rereading validated fields. These guarantees hold whether or not the platform first copies the entire buffer into private memory. Moreover, `Queue` enforces buffer ownership and `VolatileReader` prevents repeated reads of the same bytes throughout safe driver code, not just within the generated parsers. Unsafe operations are confined to the typed VirtIO abstractions and FFI validation entries, allowing Rust drivers to use the framework without `unsafe` code.

We implemented the framework using Rust for Linux and ported the virtio-rng and virtio-vsock drivers to safe Rust. We also modified the existing C virtio-net and virtio-blk drivers to use the generated validators through FFI. To assess the soundness of our typed abstractions, we implemented a Rust model of the kernel interfaces that allows Miri [13] to execute the drivers against a hostile device.

We evaluated the framework through fault injection, fuzzing, Miri-based testing, performance measurements, and an analysis of the defects found in our audit. The validators rejected malformed inputs injected by a modified QEMU, including a replay of CVE-2025-39718. In a targeted TOCTOU experiment, the Rust virtio-vsock driver prevented inconsistent reads observed in the C drivers. During 24 hours of fuzzing with 2.15 billion packets, the Rust virtio-vsock driver triggered no crashes. The Miri model identified five concurrency and lifecycle issues in our typed abstractions, all of which we fixed. The median receive throughput of the Rust virtio-vsock driver was 2.4–6.5% lower than that of the C driver under the original refill policy, but differed by at most 1.6% when both drivers used an optimized refill policy. To assess the applicability of the framework beyond the implemented drivers, we analyzed the 15 defects found

ⓒ2026 Information Processing Society of Japan

in our audit and found that its mechanisms could prevent 12 of them if the affected drivers adopted the framework with appropriate constraints. We experimentally verified protection against three of these defects using the implemented drivers.

Our main contributions are as follows:

- Typed VirtIO abstractions that enforce buffer ownership and single-fetch access to host-writable memory through safe Rust interfaces (Sections 3.1 and 3.2).
- A DSL that generates validating parsers from annotated Rust structs, supporting variable-length records, cross-field constraints, and validation against external context (Section 3.3).
- An implementation in Linux and an evaluation of its security and performance, including Miri-based testing of the typed abstractions (Sections 5.2, 5.3, and 5.4).
- An analysis of 15 defects in Linux 7.3, demonstrating the applicability and limitations of the framework beyond the implemented drivers (Section 5.5).

2. Background and Threat Model

2.1 VirtIO

VirtIO is a standard interface between guest drivers and virtual devices provided by the host [1]. Devices exchange data with drivers through *virtqueues*. In the split-ring format, each virtqueue consists of a descriptor table, an available ring written by the driver, and a used ring written by the device. To receive data, the driver posts a buffer by adding a descriptor containing its address and size to the available ring and notifying the device. The device writes into the buffer and places the descriptor's index and the number of bytes written, called the *used length*, into the used ring. The device then typically notifies the driver through an interrupt, allowing it to reclaim the buffer and its used length. Devices also expose a configuration space containing parameters such as the capacity and zone size of a block device.

Drivers read three kinds of device-supplied values: received buffer contents, used lengths, and device configuration fields. A received buffer typically begins with a header whose fields describe the remaining data.

The virtio-vsock device carries socket traffic between a guest and its host, including communication with security-sensitive components of a CVM. For example, Kata Containers uses vsock for communication between the host runtime and the agent inside the guest [14]. Intel's attestation library also communicates with a host-side quote generation service over vsock to obtain a TDX quote [15].

2.2 Confidential VMs

Intel TDX and AMD SEV-SNP protect the private memory of a CVM from access by the host. Since virtual devices provided by the host cannot access private memory, the guest uses shared memory for device I/O. Linux places virtqueue rings in shared memory and uses bounce buffers implemented by the software I/O translation lookaside buffer (swiotlb) to transfer data between private and shared memory. When a driver maps a private buffer for device access, swiotlb substitutes a shared bounce buffer. When the driver unmaps the buffer after the device has written to it, swiotlb copies the received data back into private memory. In contrast, accesses to device configuration space can obtain host-controlled values on successive reads.

2.3 Safe and Unsafe Rust

Rust distinguishes between safe and unsafe code. In safe code, the compiler enforces memory safety rules, including bounds checking, reference lifetimes, aliasing restrictions, and safe sharing of data between threads. Unsafe code can perform operations that the compiler cannot verify, such as dereferencing raw pointers and calling C functions, so developers must ensure that these operations preserve Rust's safety invariants. A safe abstraction encapsulates unsafe operations behind an interface that safe code can use. Such an abstraction is *sound* if no safe code using it can cause undefined behavior, such as memory errors or data races. Therefore, drivers written in safe Rust are free of undefined behavior as long as the abstractions they use are sound.

2.4 Threat Model

We follow the CVM threat model [2], in which the host and its virtual devices are untrusted. The host can modify any shared page at any time, including while the guest reads it. It can also choose the used length and descriptor index of each completion, return descriptors that the driver has already reclaimed, and raise virtqueue interrupts at arbitrary times, including during driver initialization and removal. Our goal is to prevent malicious device input from causing memory safety violations in Rust drivers and ensure that device-supplied fields are validated against declared constraints before use. Denial of service caused by the host withholding I/O is out of scope, since the host can prevent the guest from running altogether. Side-channel attacks are also out of scope.

We trust the guest kernel components that our drivers use, including the virtqueue implementation, swiotlb, and the virtio-vsock socket protocol layer.

Even with bounce buffering, the host can modify virtqueue rings and device configuration space between reads. Moreover, bounce buffering incurs additional CPU overhead, and Bifrost found that bounce buffering and packet processing can account for more than half of the CPU cycles spent on network I/O in CVMs [16]. Our framework therefore treats device-writable buffers as potentially mutable by the host at any time, ensuring that input validation remains safe even when the receive path bypasses bounce buffering.

3. Design

With our framework, a developer can write VirtIO drivers in safe Rust and specify the layout of each device-written structure in our DSL as a Rust struct, whose fields carry

ⓒ2026 Information Processing Society of Japan

the constraints that a valid structure satisfies. A procedural macro generates a parser from each struct at compile time. The driver posts buffers through a typed queue, reclaims them from the queue when the device has used them, and passes the device-writable part of each buffer to the parser, which returns a value if every field satisfies its declared constraints, and an error otherwise.

3.1 Typed Queues

The C virtqueue interface identifies each posted buffer by an untyped pointer, called the *token*, which is returned with the used length when the device has used the buffer. The interface leaves three obligations to the driver. First, the memory behind the token must stay allocated, and must not be reused, until the device returns it. Second, the token must be converted back to the type that was posted. Third, the calls on one queue must be serialized, because the interface updates the driver's side of the ring without a lock.

A `Queue` enforces each obligation through a type. It is a virtqueue whose tokens are owned buffers of one type, such as socket buffers, so the conversion back is safe. Its `add_inbuf` takes a buffer by value, so the caller can no longer refer to it, and its `get_buf` returns the same buffer as a value that the driver owns again, together with the used length. As a result, a buffer held by the device is not reachable from safe driver code and cannot be freed, reused or read before the device has returned it. The queue also cuts the used length to the size of the buffer for every driver, avoiding vulnerabilities like CVE-2026-64456 in the C virtio-rng driver by construction [17].

To avoid race conditions, every method that changes the state of the ring takes a mutable reference to the queue, which Rust requires to be exclusive. Since the interrupt callback receives only a shared reference to the driver's data, a queue used by both the interrupt callback and a work item has to be placed behind a lock, whose type is safe to share between CPUs.

A generic adapter handles the probe and removal of each driver and enforces the lifecycle rules. During probe, the driver's data is stored before the device is marked ready, and no buffer is posted before the data is stored. On removal, the adapter waits for running callbacks, has the driver stop posting buffers, resets the device and drops the driver's data. The virtqueues are deleted only after every queue handle has been dropped, so no handle can access a freed virtqueue. Buffers that the device never returned are then unmapped and freed through the trusted kernel interfaces.

3.2 A Forward-Only Cursor over Host-Writable Memory

The Rust compiler assumes that the bytes behind a shared reference to a byte slice (`&[u8]`) do not change while the reference exists. It may therefore read a byte a second time where the program reads it once, which reintroduces a double fetch that the source does not contain [18]. Host-writable memory violates this assumption, so creating such a reference to it is undefined behavior. Instead, a `VolatileSlice` borrows host-writable memory as a slice of opaque bytes, whose contents are not assumed to be stable, and reads them only with volatile loads. A volatile load is a single access that the compiler may neither repeat nor merge, as with `READ_ONCE` in Linux.

A `VolatileSlice` can only be read through a `VolatileReader`, a cursor that only moves forward, and creating the cursor consumes the slice. The cursor has no seek, rewind or read that leaves the position in place, and it can be neither copied nor cloned. Its single read method fetches the next N bytes into an array in private memory and advances past them, or fails if fewer than N bytes remain. A parser that holds a cursor therefore reads each byte of the shared region at most once. Moreover, the queue returns each used buffer as a value that can be read only once, through one such cursor, and safe driver code cannot create a cursor in any other way, so each byte is fetched at most once even without a parser. An opaque payload is an exception, since the driver passes it on without reading it.

3.3 A DSL for Device-Written Structures

Figure 3 shows the specification of the virtio-vsock packet. Its fixed fields mirror the C header field by field, and its last field declares the payload. The `#[derive]` line invokes our procedural macro, which generates the parser. An attribute on a field declares a constraint on its value as bounds, a set of allowed values, an enum whose variants are the allowed values, or a custom function. In the figure, `validate_cid` checks that an address fits in 32 bits, and the bound on `len` limits the payload to 64 KiB.

The payload takes one of three forms: opaque bytes whose length a field gives, as `len` does in the figure, an array of records whose count a field gives, or a sequence of records that fills the rest of the buffer, each starting with a header that gives its type and its length. Each record is itself a struct with its own specification, and a struct can also declare a *cross-field check*, a function that checks an invariant across fields after every field has passed its own check.

The generated parser fetches the fixed fields with one read of the cursor and checks their constraints in private memory. It also rejects a length field that exceeds the bytes left in the cursor, such as the payload length of CVE-2025-39718, which is below the 64 KiB bound but exceeds the buffer. The upstream fix added the same check to the C driver manually, which is generated from the specification in our framework. The parser reads the records from the same cursor in the same pass, so a count cannot change after it has bounded the records. For example, the zone report of virtio-blk lists the zones of a zoned disk as an array of zone descriptors, whose count is in its header. Its cross-field check rejects a descriptor whose start plus capacity overflows or whose write pointer lies outside the zone, and its parser rejects a report that claims more descriptors than the buffer holds,

ⓒ2026 Information Processing Society of Japan

```
#[derive(VirtioBuffer)]
#[repr(C)]
pub struct VsockPacket {
    #[validate(custom = "validate_cid")]
    pub src_cid: Le64,
    #[validate(custom = "validate_cid")]
    pub dst_cid: Le64,
    pub src_port: Le32,
    pub dst_port: Le32,
    #[validate(max = "VIRTIO_VSOCK_MAX_PKT_BUF_SIZE")]
    pub len: Le32,
    #[validate(enum = "VsockType")]
    pub type_: Le16,
    #[validate(enum = "VsockOp")]
    pub op: Le16,
    pub flags: Le32,
    pub buf_alloc: Le32,
    pub fwd_cnt: Le32,
    #[virtio_payload(len = "len")]
    pub payload: Payload<u8>,
}
```

Fig. 3 The specification of the virtio-vsock packet, where `Le16`, `Le32` and `Le64` are little-endian integers. The generated parser reads the 44-byte header once, checks the five constrained fields, and rejects a payload that exceeds the buffer.

which caused an out-of-bounds read in the C driver before Linux 7.3.

A specification can also declare a *context*, a value from outside the buffer that its parser passes to the custom and cross-field checks. For example, the zone report takes the zone size from the configuration space, so its check also rejects a descriptor whose capacity exceeds the zone size. Similarly, the device statistics reply of virtio-net is a sequence of records, each holding the counters of one queue and one type of statistics, and it takes the request that it answers as its context. The C driver trusts the queue and the type of each record, so a record for a nonexistent queue causes an out-of-bounds write to its output array, whereas the check accepts only a queue and a type listed in the request.

C drivers call the same parsers through FFI, where each *validation entry* is a Rust function with a C signature that wraps a pointer and a length in a `VolatileSlice` and runs the parser. We call a parser together with its entry a *validator*. A validation entry returns checked values, such as the payload length of a packet or a private copy of each zone descriptor. To preserve the single-fetch guarantee, the C driver must use these returned values without rereading the corresponding fields from shared memory.

3.4 Security Analysis

Our framework protects Rust drivers against the capabilities of a malicious host defined in the threat model, assuming that the typed abstractions and trusted kernel interfaces are sound. Although the host can rewrite any byte in host-writable memory at any time, changes to bytes already fetched by the cursor cannot affect the values copied into private memory. The queue clamps a used length that exceeds the buffer size, preventing the parser from reading beyond the buffer. Device configuration values are also validated by the generated parsers. The trusted virtqueue implementation rejects descriptor indices outside the ring and completions for buffers that are no longer posted. Since each token is returned only once, each buffer is reconstructed from its token exactly once and cannot have multiple owners. Interrupt callbacks access queues through locks, while the adapter prevents callbacks from accessing driver data before initialization or after removal. Thus, assuming sound abstractions and trusted kernel interfaces, malicious device input cannot cause undefined behavior in safe Rust driver code. The generated parsers additionally enforce declared input constraints, but the developer remains responsible for specifying constraints required by the device protocol. Existing C drivers benefit from the generated validators only where they use the validated values without rereading shared memory.

Table 1 Code added to Linux 7.3 by layer. Lines exclude blank and comment lines. The `unsafe` column counts `unsafe` blocks in the source.

Layer	Lines	`unsafe`
Typed VirtIO abstractions	946	116
DSL macro and runtime	868	0
Rust drivers and device specifications	1,051	0
C changes and helpers	475	–
FFI validation entries	119	9
Miri model	1,958	114
Unit and fuzz tests	1,783	31

4. Implementation

We implemented the framework in Linux 7.3 using Rust for Linux [10]. Table 1 summarizes the code we added. In addition to the VirtIO primitives, the typed VirtIO abstractions include bindings for socket buffers, the vsock transport, work items, completions, and the hardware random number generator core. Among the production components, only the typed VirtIO abstractions and FFI validation entries contain `unsafe` code, primarily for calling C functions and dereferencing raw pointers.

We ported virtio-rng to safe Rust, registering it with the hardware random number generator core and posting one buffer for each request from the core.

We also ported virtio-vsock to safe Rust, managing its receive, transmit, and event queues while reusing the existing C socket protocol layer. On the receive side, a work item drains the receive queue, validates each packet using the generated parser, and passes it to the protocol layer. Since the C protocol layer reads the packet header from the socket buffer, leaving the original header in host-writable memory could reintroduce a double-fetch vulnerability. To prevent this, the driver reserves space for a header in private memory at the beginning of the socket buffer, outside the region mapped to the device. The driver copies the validated header into this reserved space so that the protocol layer uses the checked values rather than the original header.

The C virtio-blk driver uses the generated validators for three types of device input: the status byte of completed requests, zone reports, and device configuration space. Simi-

ⓒ2026 Information Processing Society of Japan

larly, the C virtio-net driver validates device statistics replies before processing their records.

5. Evaluation

5.1 Setup

All experiments ran on a host with an Intel Core i9-13900K running Linux 6.18 and QEMU 11.1 with KVM. The host does not support TDX, so the guest runs without memory encryption or bounce buffers. The guest kernel is Linux 7.3 with our changes, built with Clang 23.1 and rustc 1.99. The performance experiments use stock QEMU, with the host kernel's vhost-vsock worker handling packet transfers.

We compare five variants of the virtio-vsock receive path. The *vanilla C* driver is the upstream driver, and the *hardened C* driver is the same driver calling the validation entry of virtio-vsock through FFI. The *Rust driver* is our port. In the performance experiments, these three refill the receive queue after each drain, as the upstream driver does. To compare the drivers under a second refill policy, we also modified the vanilla C driver and the Rust driver to refill the queue inside the drain, called the *drain-refill C* and *drain-refill Rust* drivers. All five allocate receive buffers of the same size and use the same ring of 128 buffers and the same suppression of callbacks during the drain, and they share the rest of the kernel. For the replays of CVE-2025-39718, we also built the C driver as it was before the upstream fix. For input injection, we modified QEMU to emulate devices in user space and manipulate device-supplied data. We enabled the Kernel Address Sanitizer (KASAN) in the guest for the fuzzing, zone report, and statistics reply experiments to detect invalid memory accesses.

5.2 Robustness Against a Hostile Device

We injected malformed data from the host into the drivers and checked the guest for memory errors and crashes.

Replay of CVE-2025-39718

The modified QEMU delivered 16 packets with a valid header whose payload length was 60,000 bytes, in buffers of about 3.7 KiB. The C driver without the upstream fix panicked in `skb_put` on the first such packet, whereas the Rust driver and the hardened C driver rejected all 16 packets and kept running.

Zone Report Injection

For virtio-blk, the modified QEMU forwarded and corrupted the zone reports of a zoned disk on the host. First, we made the reports claim more descriptors than the buffer could hold. In two boots of the C driver without the validator or the upstream fix, KASAN reported an out-of-bounds read both times. In three boots with the validator, the driver rejected the report before it read a descriptor. Then we corrupted one field of the first descriptor in each boot, and the validator rejected an unknown zone type, an unassigned zone state, a write pointer below and one above the zone, a capacity larger than the zone, and a fully random descriptor, while KASAN reported no memory error.

Statistics Reply Injection

For virtio-net, the modified QEMU offered device statistics and answered every request of the guest with the same reply in each of three boots. The first reply held a record for a nonexistent receive queue, the second a record whose length was zero, and the third had a used length one record shorter than the reply. Without the validator, KASAN reported an out-of-bounds write for the first reply. The read of the statistics hung indefinitely for the second, and the driver returned the counters of the record past the used length for the third. With the validator, the driver rejected the first two replies with no KASAN report, and returned only the counters of the records within the used length for the third.

TOCTOU Race

The modified QEMU delivered valid packets with a 64-byte payload one at a time and, for 1 ms after each delivery, alternated the payload length in the header between 64 and 60,000 bytes. An instrumented protocol layer compared the payload length in each header with the socket buffer length set by the driver from the validated value. Of 28,672 packets, the lengths differed in 267 with the vanilla C driver and 168 with the hardened C driver, both of which leave the header in the device's buffer. In contrast, no mismatches occurred with the Rust driver, which passes a validated copy of the header to the protocol layer. These results show that validation alone does not prevent TOCTOU vulnerabilities if the driver subsequently rereads host-writable memory.

Fuzzing

In fuzzing mode, the modified QEMU continuously fills every posted receive buffer. Three packets in four belong to connections to a server in the guest. QEMU drives these connections through the states of the vsock protocol by following the replies of the guest, and mutates one header field in one of every eight of their packets. The fourth packet is a single packet with mutated or random bytes. In addition, the used length of one packet in sixteen is wrong, and random events are pushed on the event queue. During 24 hours of this injection against a KASAN kernel, QEMU injected 2.15 billion packets, and the Rust virtio-vsock driver triggered no crash or KASAN report. In contrast, the same injection against the C driver without the upstream fix panicked the guest in `skb_put` 0.8 seconds after boot.

5.3 Overhead

We measured the receive throughput, the CPU time and the round-trip latency of the five variants. A host program sent 8 GiB in 64 KiB messages, 2 GiB in 4 KiB messages and 128 MiB in 256-byte messages over one vsock stream to the guest. The guest had four vCPUs pinned to distinct performance cores, and the host program and the vhost worker to two more. In each of the six rounds, the five variants were booted back to back and each was measured three times after a warm-up, and the median of the three was taken.

Overhead of virtio-vsock

The receive throughput of the Rust driver was 2.4–6.5% lower than that of the vanilla C driver, while that of the

ⓒ2026 Information Processing Society of Japan

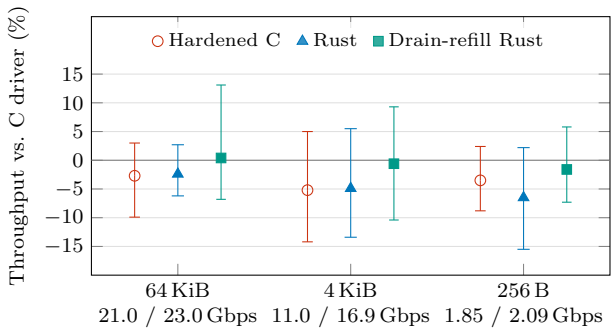

Fig. 4 Receive throughput of each variant relative to the C driver with the same refill policy in the same round, by message size, which is the vanilla C driver for the hardened C and Rust drivers and the drain-refill C driver for the drain-refill Rust driver. A mark is the median over six rounds, and its line spans the lowest and the highest round. The labels give the median throughput of the vanilla C driver and of the drain-refill C driver.

hardened C driver was 2.7–5.2% lower, as shown in Figure 4. With the drain-refill policy, the Rust driver's throughput ranged from 1.6% lower to 0.4% higher than that of the C driver, with differences smaller than the variation across rounds. The guest's CPU time per transfer was 3.5–7.6% higher for the Rust driver and 3.3–8.7% higher for the hardened C driver. The median round-trip latency of a 64-byte message was 9.5–10.5 µs, with no consistent difference among the five variants.

The virtio-blk Validator

To measure the cost of the validation entry for the C virtio-blk driver, we booted the kernel with and without the virtio-blk validators in six rounds. In each boot, fio 3.38 read a disk that QEMU emulates without host I/O, issuing 4 KiB random reads through io_uring from four jobs with 32 requests in flight each. With the validator, the guest's CPU time per read was between 4.8% lower and 5.8% higher than without it, without a consistent sign across rounds. The median read rate was 0.3% lower with the validator, indicating no measurable throughput overhead under this workload.

5.4 Issues Found by the Miri Model

We used Miri to detect potential soundness violations in the typed abstractions by executing the two Rust drivers against a model of their kernel environment. Miri is an interpreter that detects undefined behavior in Rust programs, including memory errors and data races [13]. Since it cannot run C code, we implemented the kernel interfaces called by the drivers as a Rust model of functions with C linkage. A test binds a driver to a model device emulated by a separate thread. As a hostile host, this thread can claim more bytes than a buffer holds, return a descriptor twice, and raise interrupts during probe and removal. The model device writes a buffer only before returning it to the driver, so the tests do not cover a host that changes a buffer while the driver reads it. Miri runs the tests under 64 random thread schedules, where each thread in the model stands for a CPU. The model checks the rules required by the kernel interfaces, such as the lockdep rules and the lifecycle of the virtqueues. It also makes the allocations of the probe fail in turn and checks that every allocation is freed when a driver is unbound or its probe fails, so that every error path of the probe is covered.

The Miri model found five issues in our typed abstractions. First, our queue interface allowed two CPUs to change one virtqueue concurrently, which the C virtqueue forbids, and Miri reported it as a data race. The receive work of virtio-vsock raced its probe this way, and the interrupt callback of virtio-rng raced its read path when the device returned a descriptor twice. The same race was also reproduced in the virtio-rng driver of mainline Linux. We fixed it by making every method that changes a virtqueue take a mutable reference. Second, the queue interface allowed a driver to post a buffer that it was still reading, so virtio-rng gave its buffer back to the device before it had copied the random bytes out when the device answered two requests consecutively. We fixed it by making the queue take a buffer by value. Third, the adapter never deleted the virtqueues at removal or after a failed probe, which the model's check of the virtqueue lifecycle caught. Fourth, it left the buffers on a virtqueue to the driver at removal, so virtio-vsock leaked its posted buffers, which the model found unfreed. Fifth, it left the reset of the device to the driver, so at removal virtio-vsock reset its device before the protocol layer stopped posting packets to it, and the model reported a packet posted to the reset device. We fixed the last three by making the adapter delete the virtqueues, free their buffers, and reset the device only after the driver has stopped posting to it. After the fixes, both drivers passed every test under all the sampled schedules.

5.5 Defects in the Mainline Drivers

To assess the applicability of our framework beyond the implemented drivers, we conducted an LLM-assisted audit of the VirtIO drivers in mainline Linux 7.3. We used Claude to identify potential defects involving insufficient validation of device-supplied input and reproduced the identified defects to confirm their impact. The audit identified 15 defects in 12 drivers, as listed in Table 2, of which 11 had no complete fix posted before our audit. We have submitted patches for the virtio-net and rpmsg defects [19], [20], both of which have been confirmed by their maintainers. We are preparing fixes for the remaining defects, along with two races involving virtio-rng [21].

As Table 2 shows, the framework's mechanisms could prevent 12 of the 15 defects if the affected drivers adopted the framework with appropriate constraints. Ten of these defects would require at most one additional DSL constraint on a field. For the three defects in virtio-net and virtio-blk, the implemented validators rejected the malformed input under injection. For virtio-blk, the modified QEMU supplied invalid block exponent and write cache mode values through the configuration space, which the driver rejected during probe. The remaining three defects, involving 9p zero-copy, virtio-fs, and virtio-iommu, require two extensions to the

ⓒ2026 Information Processing Society of Japan

Table 2 Defects in the mainline VirtIO drivers and the mechanisms of the framework that cover them. Effect: M, memory corruption; D, a guest crash or hang; U, other undefined behavior (an oversized shift). *: no complete fix posted before our audit.

Driver	Device input	Eff.	Mechanism
9p	config tag length	M	field constraint
9p*	zero-copy used length	M	extension
gpio-virtio*	config names size	M	field constraint
rpmsg*	used, message length	M	queue, payload
rpmsg*	config buffer sizes	M	field constraint
virtio-blk*	config block exponent	U	field constraint
virtio-blk*	config write cache	D	field constraint
virtio-console*	short used length	D	cursor
virtio-fs	reply error, length	M	field constraint, extension
virtio-input*	config bitmap size	D	field constraint
virtio-iommu	property length	M	extension
virtio-mem	config block size	D	field constraint
virtio-net*	stats reply records	M	records, context
virtio-scsi*	config `seg_max`	M	field constraint
virtio-snd*	completion at removal	M	adapter

framework, as discussed in Section 7.

6. Related Work

Hardening drivers against the device.

Intel's hardening effort for Linux TDX guests identifies host input through static analysis and restricts the guest to audited drivers [8]. A check of the used length in the VirtIO core has been proposed [22], but it covers only one of the three kinds of device-supplied values, since the core cannot validate buffer contents whose layout is known only to the driver. Gramine-TDX replaces the guest kernel with a small library OS [23]. In contrast, our framework retains the Linux kernel and supports both safe Rust drivers and incremental hardening of existing C drivers. Our typed abstractions enforce safe access to device-writable memory, while the DSL centralizes the validation constraints for device-specific structures.

Validated parsers.

Nail generates parsers and generators for data formats from a grammar [24]. Vest provides verified parser combinators for Rust [25]. AVID derives parsers from Rust struct definitions using procedural macros, but its parsing expression grammars cannot express layouts in which a length field determines how the remaining input is parsed [26]. In the Rust ecosystem, `untrusted` provides forward-only parsing of byte slices [27], while `zerocopy` derives checks for the validity of byte representations, such as enum discriminants [28]. However, `zerocopy` does not directly express device-specific constraints such as value ranges, field-dependent lengths, or arrays with device-supplied counts, and its byte-slice interface cannot safely access host-writable memory. EverParse3D generates C validators that are formally verified to be memory-safe and free of double fetches using F*, and has been deployed in Hyper-V for network virtualization messages [12]. However, these guarantees apply to the validators and do not prevent their callers from rereading shared memory or accessing buffers still owned by the device. In contrast, our framework combines generated validators with typed abstractions that enforce buffer ownership and single-fetch access throughout safe Rust driver code.

Rust drivers and kernels.

Levy et al. observed that Rust lifetimes cannot express the duration of a DMA transfer and assigned static lifetimes to DMA buffers [29]. Our typed queues address this problem by transferring buffer ownership to the queue for the duration of the transfer. Asterinas exposes device-writable memory as untyped memory, which kernel code accesses as plain data, and uses the IOMMU to restrict device writes [30]. Its threat model focuses on preventing devices from corrupting kernel memory, whereas ours focuses on malicious inputs from host-controlled virtual devices. A study of Rust for Linux found soundness bugs in its safe abstractions [31], motivating our Miri-based testing. The closest work in Linux is a proposal for Rust VirtIO bindings [32], which also wraps virtqueues and transfers buffer ownership. However, its authors acknowledge that virtqueue lifetimes require unsafe pointer dereferences, leaving room for misuse. Its virtqueue interface also returns device-supplied used lengths without bounding them by buffer sizes, leaving validation to individual drivers. Our abstractions enforce these properties through safe interfaces.

Checking drivers against a model of their environment.

SDV and DDVerify statically check C drivers for Windows and Linux, respectively, against hand-written models of operating system interfaces, with DDVerify also modeling concurrent interrupts [33], [34]. KernMiri extends Miri with physical memory and page tables to check `unsafe` code in Asterinas [30]. XRust executes Rust for Linux drivers in user space against models of their kernel dependencies and detects logical races by comparing concurrent executions with serial ones [35]. Unlike KernMiri, our approach leaves Miri unchanged and implements the kernel interfaces in Rust. Our model checks undefined behavior and kernel interface invariants together, while also simulating malicious device behavior and allocation failures.

7. Discussion

Soundness of the abstractions.

Miri checks the typed abstractions only on executed paths, under sampled thread schedules, and against the invariants encoded in our model. A soundness proof in the style of RustBelt [36] would cover all execution paths but require formal specifications of the kernel interfaces used by the abstractions, as in Veld's verification of Linux drivers using Verus [37].

Layouts that depend on features.

The DSL allows external context values to influence validation constraints, but not the layout of a parsed structure. For example, the virtio-net receive header contains fields whose presence depends on negotiated features, so porting its data path would require context-dependent layouts, as supported by the parameterized types of EverParse3D and Nail [12], [24].

ⓒ2026 Information Processing Society of Japan

Layouts beyond one buffer.

The cursor currently supports only contiguous buffers, and record lengths must include their headers. These restrictions leave three defects found in our audit outside the framework's current coverage. For 9p zero-copy replies and virtio-fs requests with page arguments, a single reply can span multiple buffers, requiring a cursor that supports buffer chains. The probe properties of virtio-iommu require a different record layout, where lengths exclude headers, unknown types have opaque bodies, and a special type terminates the sequence.

Scope of the ports.

Our Rust virtio-vsock driver reuses the existing C socket protocol layer, which processes packets after validation. The port therefore relies on the protocol layer as a trusted component, since porting it to safe Rust would require additional Rust for Linux bindings. Similarly, we retained virtio-blk and virtio-net in C and integrated the generated validators through FFI. The Rust virtio-vsock driver also copies the validated header into a reserved region of the socket buffer for the C protocol layer, adding some overhead.

8. Conclusion

We presented a framework that uses Rust's type system to harden VirtIO drivers against malicious hosts in confidential virtual machines (CVMs). The framework combines typed VirtIO abstractions that enforce buffer ownership and single-fetch access with a domain-specific language (DSL) that generates input validators. We implemented the framework in Linux, ported virtio-rng and virtio-vsock to safe Rust, and integrated the generated validators into the C virtio-net and virtio-blk drivers. The validators rejected malformed device input, while the Rust virtio-vsock driver prevented inconsistent reads in a targeted TOCTOU experiment and triggered no crashes during 24 hours of fuzzing. Its median receive throughput was 2.4–6.5% lower than that of the C driver under the original refill policy, but differed by at most 1.6% when both drivers used an optimized policy. An LLM-assisted audit identified 15 defects in 12 Linux VirtIO drivers, 12 of which our framework could prevent with appropriate constraints, including three for which we experimentally validated protection. Our Miri model identified five concurrency and lifecycle issues in the typed abstractions, all of which we fixed.

In future work, we plan to verify the soundness of the typed abstractions against specifications of the kernel interfaces and explore more thread schedules using loom [38]. We also plan to extend the DSL to support feature-dependent and multi-buffer layouts and evaluate the drivers on TDX or SEV-SNP hardware with memory encryption and bounce buffers.

References

[1] OASIS Open: *Virtual I/O Device (VIRTIO) Version 1.3* (2023).

[2] Reshetova, E. and Bilbao, C.: Confidential Computing in Linux for x86 Virtualization, Linux kernel documentation (online), available from ⟨`https://docs.kernel.org/6.8/security/snp-tdx-threat-model.html`⟩ (accessed 2026-10-21).

[3] NIST: CVE-2025-39718: vsock/virtio: Validate length in packet header before skb_put(), National Vulnerability Database (online), available from ⟨`https://nvd.nist.gov/vuln/detail/cve-2025-39718`⟩ (accessed 2026-10-21).

[4] NIST: CVE-2025-40292: virtio-net: fix received length check in big packets, National Vulnerability Database (online), available from ⟨`https://nvd.nist.gov/vuln/detail/cve-2025-40292`⟩ (accessed 2026-10-21).

[5] NIST: CVE-2026-64552: virtio-net: fix len check in receive_big(), National Vulnerability Database (online), available from ⟨`https://nvd.nist.gov/vuln/detail/cve-2026-64552`⟩ (accessed 2026-10-21).

[6] Hetzelt, F., Radev, M., Buhren, R., Morbitzer, M. and Seifert, J.-P.: VIA: Analyzing Device Interfaces of Protected Virtual Machines, *Proc. ACSAC '21*, pp. 273–284 (online), DOI: 10.1145/3485832.3488011 (2021).

[7] Wang, P., Krinke, J., Lu, K., Li, G. and Dodier-Lazaro, S.: How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel, *Proc. USENIX Security '17*, pp. 1–16 (2017).

[8] Intel Corporation: Intel Trust Domain Extension Guest Linux Kernel Hardening Strategy, intel.github.io (online), available from ⟨`https://intel.github.io/ccc-linux-guest-hardening-docs/tdx-guest-hardening.html`⟩ (accessed 2026-10-21).

[9] Lefeuvre, H., Chisnall, D., Kogias, M. and Olivier, P.: Towards (Really) Safe and Fast Confidential I/O, *Proc. HotOS '23*, pp. 214–222 (online), DOI: 10.1145/3593856.3595913 (2023).

[10] Rust for Linux contributors: Rust for Linux, rust-for-linux.com (online), available from ⟨`https://rust-for-linux.com/`⟩ (accessed 2026-10-21).

[11] Li, Z., Narayanan, V., Chen, X., Zhang, J. and Burtsev, A.: Rust for Linux: Understanding the Security Impact of Rust in the Linux Kernel, *Proc. ACSAC '24*, pp. 548–562 (online), DOI: 10.1109/ACSAC63791.2024.00054 (2024).

[12] Swamy, N., Ramananandro, T., Rastogi, A., Spiridonova, I., Ni, H., Malloy, D., Vazquez, J., Tang, M., Cardona, O. and Gupta, A.: Hardening Attack Surfaces with Formally Proven Binary Format Parsers, *Proc. PLDI '22*, pp. 31–45 (online), DOI: 10.1145/3519939.3523708 (2022).

[13] Jung, R., Kimock, B., Poveda, C., Sánchez Muñoz, E., Scherer, O. and Wang, Q.: Miri: Practical Undefined Behavior Detection for Rust, *Proc. ACM Program. Lang.*, Vol. 10, No. POPL, pp. 1383–1411 (online), DOI: 10.1145/3776690 (2026).

[14] Kata Containers contributors: Kata Containers, GitHub (online), available from ⟨`https://github.com/kata-containers/kata-containers`⟩ (accessed 2026-10-21).

[15] Intel Corporation: Intel Data Center Attestation Primitives (Intel DCAP), GitHub (online), available from ⟨`https://github.com/intel/confidential-computing.tee.dcap`⟩ (accessed 2026-10-21).

[16] Li, D., Mi, Z., Ji, C., Tan, Y., Zang, B., Guan, H. and Chen, H.: Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines, *Proc. USENIX ATC '23*, pp. 1–15 (2023).

[17] NIST: CVE-2026-64456: hwrng: virtio: clamp device-reported used.len at copy_data(), National Vulnerability Database (online), available from ⟨`https://nvd.nist.gov/vuln/detail/cve-2026-64456`⟩ (accessed 2026-10-21).

[18] Xen Project: XSA-155: paravirtualized drivers incautious about shared memory contents, Xen Security Advisories (online), available from ⟨`https://xenbits.xen.org/xsa/advisory-155.html`⟩ (accessed 2026-10-21).

[19] Wang, L.: [PATCH net] virtio_net: reject stats replies for queues outside the request, netdev mailing list (online), available from ⟨`https://lore.kernel.org/all/20261022182710.2055771-1-karin0.zst@gmail.com/`⟩ (accessed 2026-10-28).

[20] Wang, L.: [PATCH] rpmsg: virtio: bound the buffer sizes read from config space, linux-remoteproc mailing list (online), available from ⟨`https://lore.kernel.org/all/20261022194129.2233880-1-karin0.zst@gmail.com/`⟩ (accessed 2026-10-28).

[21] NIST: CVE-2026-45949: hwrng: core - use RCU and work_struct to fix race condition, National Vulnerability Database (online), available from ⟨`https://nvd.nist.gov/`

ⓒ2026 Information Processing Society of Japan

vuln/detail/cve-2026-45949⟩ (accessed 2026-10-28).

[22] Wang, J.: virtio_ring: validate used buffer length, version 5, LKML (online), available from ⟨https://lkml.iu.edu/hypermail/linux/kernel/2110.3/03750.html⟩ (accessed 2026-10-21).

[23] Kuvaiskii, D., Stavrakakis, D., Qin, K., Xing, C., Bhatotia, P. and Vij, M.: Gramine-TDX: A Lightweight OS Kernel for Confidential VMs, *Proc. ACM CCS '24*, pp. 4598–4612 (online), DOI: 10.1145/3658644.3690323 (2024).

[24] Bangert, J. and Zeldovich, N.: Nail: A Practical Tool for Parsing and Generating Data Formats, *Proc. OSDI '14*, pp. 615–628 (2014).

[25] Cai, Y., Singh, P., Lin, Z., Bosamiya, J., Gancher, J., Surbatovich, M. and Parno, B.: Vest: Verified, Secure, High-Performance Parsing and Serialization for Rust, *Proc. USENIX Security '25*, pp. 6917–6935 (2025).

[26] Dawson, M. D.: Secure Input Validation in Rust with Parsing-Expression Grammars, Master's thesis, Massachusetts Institute of Technology (2019).

[27] Smith, B.: untrusted: Safe, fast, zero-panic, zero-crashing, zero-allocation parsing of untrusted inputs in Rust, docs.rs (online), available from ⟨https://docs.rs/untrusted⟩ (accessed 2026-10-21).

[28] zerocopy contributors: zerocopy::TryFromBytes, Linux kernel rustdoc (online), available from ⟨https://www.kernel.org/doc/rustdoc/latest/zerocopy/trait.TryFromBytes.html⟩ (accessed 2026-10-21).

[29] Levy, A., Campbell, B., Ghena, B., Pannuto, P., Dutta, P. and Levis, P.: The Case for Writing a Kernel in Rust, *Proc. APSys '17*, pp. 1–7 (online), DOI: 10.1145/3124680.3124717 (2017).

[30] Peng, Y., Tian, H., Zhang, J., Li, R., Chen, C., Jiang, J., Xian, J., Wang, X., Xu, C., Zhou, D., Luo, Y., Yan, S. and Zhang, Y.: Asterinas: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB, *Proc. USENIX ATC '25*, pp. 307–323 (2025).

[31] Li, H., Guo, L., Yang, Y., Wang, S. and Xu, M.: An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise, *Proc. USENIX ATC '24*, pp. 425–443 (2024).

[32] Pitsidianakis, M.: [PATCH RFC v3 0/6] Add Rust virtio bindings and sample device, LKML (online), available from ⟨https://lkml.iu.edu/2605.1/03063.html⟩ (accessed 2026-10-21).

[33] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K. and Ustuner, A.: Thorough Static Analysis of Device Drivers, *Proc. EuroSys '06*, pp. 73–85 (online), DOI: 10.1145/1217935.1217943 (2006).

[34] Witkowski, T., Blanc, N., Kroening, D. and Weissenbacher, G.: Model Checking Concurrent Linux Device Drivers, *Proc. ASE '07*, pp. 501–504 (online), DOI: 10.1145/1321631.1321719 (2007).

[35] Xu, Q., Sheng, Z., Chen, Z., Manikandan, A. and Huang, J.: XRust: Detecting Logical Races in Rust Linux Drivers, arXiv:2610.27325 (2026).

[36] Jung, R., Jourdan, J.-H., Krebbers, R. and Dreyer, D.: RustBelt: Securing the Foundations of the Rust Programming Language, *Proc. ACM Program. Lang.*, Vol. 2, No. POPL, pp. 66:1–66:34 (online), DOI: 10.1145/3158154 (2018).

[37] Chen, X., Li, Z., Zhang, J. and Burtsev, A.: Veld: Verified Linux Drivers, *Proc. KISV '24*, pp. 23–30 (online), DOI: 10.1145/3698576.3698766 (2024).

[38] Tokio contributors: loom: Concurrency Permutation Testing Tool for Rust, github.com (online), available from ⟨https://github.com/tokio-rs/loom⟩ (accessed 2026-10-29).

ⓒ2026 Information Processing Society of Japan