Tolerating Malicious Device Drivers in Linux

Silas Boyd-Wickizer and Nickolai Zeldovich
MIT CSAIL

ABSTRACT

This paper presents SUD, a system for running existing Linux device drivers as untrusted user-space processes. Even if the device driver is controlled by a malicious adversary, it cannot compromise the rest of the system. One significant challenge of fully isolating a driver is to confine the actions of its hardware device. SUD relies on IOMMU hardware, PCI express bridges, and message-signaled interrupts to confine hardware devices. SUD runs unmodified Linux device drivers, by emulating a Linux kernel environment in user-space. A prototype of SUD runs drivers for Gigabit Ethernet, 802.11 wireless, sound cards, USB host controllers, and USB devices, and it is easy to add a new device class. SUD achieves the same performance as an in-kernel driver on networking benchmarks, and can saturate a Gigabit Ethernet link. SUD incurs a CPU overhead comparable to existing runtime driver isolation techniques, while providing much stronger isolation guarantees for untrusted drivers. Finally, SUD requires minimal changes to the kernel—just two kernel modules comprising 4,000 lines of code—which may at last allow the adoption of these ideas in practice.

1 INTRODUCTION

Device drivers are a significant source of bugs in an operating system kernel [11, 13]. Drivers must implement complex features, such as wireless drivers running the 802.11 protocol, the Linux DRM graphics subsystem supporting OpenGL operations, or the Linux X server running with direct access to the underlying hardware. Drivers must correctly handle any error conditions that may arise at runtime [19]. Finally, drivers must execute in restrictive kernel environments, such as when interrupts are disabled, without relying on commonly-available services like memory allocation. The result is kernel crashes due to bugs in drivers, and even security vulnerabilities that can be exploited by attackers [1, 5].

Significant work has been done on trying to isolate device drivers, and to make operating systems reliable in the face of device driver failures [4, 6, 7, 10, 12, 14, 15, 21–23, 26–29, 33–36]. Many research operating systems include support to fully isolate device drivers [2, 15]. Unfortunately, work on commodity operating systems, like Linux, focuses on fault isolation to prevent common device driver bugs, and cannot protect the rest of the system from malicious device drivers [7, 30]. For instance, many driver isolation techniques *trust* the driver not to subvert the isolation, or not to livelock the system. If attackers exploit a bug in the device driver [1, 5], they can proceed to subvert the isolation mechanism and compromise the entire system. While some systems can provide stronger guarantees [28, 33], they rely on the availability of a fully-trusted, precise specification of the hardware device's behavior, which is rarely available for devices today.

This paper presents the design and implementation of SUD, a kernel framework that provides complete isolation for *untrusted* device drivers in Linux, without the need for any special programming languages or specifications. SUD leverages recent hardware support to implement general-purpose mechanisms that ensure a misbehaving driver, and the hardware device that it manages, cannot compromise the rest of the system. SUD allows unmodified Linux device drivers to execute in untrusted user-space processes, thereby limiting the effects of bugs or security vulnerabilities in existing device drivers. SUD also ensures the safety of applications with direct access to hardware, such as the Linux X server, which today can corrupt the rest of the system.

Moving device drivers to untrusted user-space code in any system requires addressing three key challenges. First, many device drivers require access to privileged CPU instructions in order to interact with its device. However, to protect the rest of the system, we need to confine the driver's execution. Second, the hardware device needs to perform a range of low-level operations, such as reading and writing physical memory via DMA and signaling interrupts. However, to protect the rest of the system from operations that a malicious driver might request of the device, we must also control the operations that can be performed by the device. Finally, we would like to reuse existing driver code for untrusted device drivers. However, existing drivers rely on a range of kernel facilities not available in user-space applications making it difficult to reuse them as-is.

SUD's design addresses these challenges for Linux as follows. First, to isolate driver code, SUD uses existing Unix protection mechanisms to confine drivers, by running each driver in a separate process under a separate Unix user ID. To provide the device driver with access to its hardware device, the kernel provides direct access to memory-mapped device IO registers using page tables, and uses other *x*86 mechanisms to allow controlled access

to the IO-space registers on the device. The kernel also provides special device files for safely managing device state that cannot be directly exposed at the hardware level.

Addressing the second challenge of confining the physical hardware managed by a driver is more difficult, due to the wide range of low-level operations that hardware devices can perform. SUD assumes that, unlike the device *driver*, the device *hardware* is trusted, and in particular, that it correctly implements the PCI express specification [25]. Given that assumption, SUD relies on an IOMMU [3, 17] and transaction filtering in PCI express bridges [25] to control the memory operations issued by the device under the control of the driver. SUD also relies on message-signaled interrupts [24] to route and mask interrupts from the device.

Finally, to support unmodified Linux device drivers, SUD emulates the kernel runtime environment in an untrusted user-space process. SUD relies on UML [9] to implement the bulk of the kernel facilities, and allows drivers to access the underlying devices by using SUD's direct hardware access mechanisms provided by the underlying kernel. The underlying kernel, in turn, exposes an upcall interface that allows the user-space process to provide functionality to the rest of the system by implementing the device driver API.

We have implemented a prototype of SUD for the Linux kernel, and have used it to run untrusted device drivers for Gigabit Ethernet cards, 802.11 wireless cards, sound cards, and USB host controllers and devices. SUD runs existing Linux device drivers without requiring any source-code modifications. A benchmark measuring streaming network throughput achieves the same performance with both in-kernel Linux drivers and the same drivers running in user-space with SUD, saturating a Gigabit Ethernet link, although SUD imposes an 8–30% CPU overhead. Our experiments suggest that SUD protects the system from malicious device drivers, even if the device driver attempts to issue arbitrary DMA requests and interrupts from its hardware device.

SUD provides complete isolation of untrusted user-space processes with access to arbitrary PCI devices, without relying on any specialized languages or specifications. SUD demonstrates how this support can be used to run unmodified Linux device drivers in an untrusted user-space process with CPU overheads comparable to other driver isolation techniques, and the same mechanisms can be used to safely run other applications that require direct access to hardware devices. Finally, we are hopeful that by making only minimal changes to the Linux kernel—two loadable kernel modules—SUD can finally put these research ideas to use in practice.

The rest of this paper is structured as follows. We first review related work in Section 2. We present the design of SUD in Section 3, and describe our implementation of SUD for the Linux kernel in Section 4. Section 5 evaluates the performance of our SUD prototype. Section 6 discusses the limitations of SUD and future work, and Section 7 concludes.

2 RELATED WORK

There is a significant amount of related work on improving the reliability, safety, and reuse of device drivers. The key focus of SUD is providing strong confinement of unmodified device drivers on Linux. In contrast, many prior systems have required adopting either a new OS kernel, new drivers, or new specifications for devices. On the other hand, techniques that focus on device driver reliability and reuse are complementary to SUD, and would apply equally well to SUD's untrusted drivers. The rest of this section surveys the key papers in this area.

Nooks [30] was one of the first systems to recognize that many kernel crashes are caused by faulty device drivers. Nooks used page table permissions to limit the effects of a buggy device driver's code on the rest of the kernel. However, Nooks was not able to protect the kernel from all possible bugs in a device driver, let alone malicious device driver code, whereas SUD can.

A few techniques for isolating kernel code at a finer granularity and with lower overheads than page tables have been proposed, including software fault isolation [6, 10] and Mondrian memory protection [34, 35]. While both SFI and MMP are helpful in confining the actions of driver code, they cannot confine the operations performed by the hardware device on behalf of the device driver, which is one of the key advantages of SUD.

Microkernels partition kernel services, including drivers, into separate processes or protection domains [2, 31]. Several microkernels, such as Minix 3 and L4, recently added support for IOMMU-based isolation of device DMA, which can prevent malicious device drivers from compromising the rest of the system [2, 15]. SUD borrows techniques from this previous work, but differs in that it aims to isolate unmodified Linux device drivers. We see this as a distinct challenge from previous work, because Linux device drivers are typically more complex than their microkernel counterparts[1] and SUD does not change the kernel-driver interface to be more amendable to isolation.

Virtual machine monitors must deal with similar issues to allow guest OS device drivers to directly access underlying hardware devices, and indeed virtualization is the key reason for the availability of IOMMU hardware, which has now been used in a number of VMMs [4, 23, 32]. In a virtual machine, however, malicious drivers can compromise their own guest OS and any applications the guest OS is running. SUD runs a separate UML process for

[1] For example, the Linux e1000 Ethernet device driver is 13,000 lines of C code, and the Minix 3 e1000 driver is only 1,250 lines.

each device driver; in this model, a driver compromising its user-space UML kernel is similar to a process compromising its libc. Thus, in SUD, the Linux kernel prevents a malicious driver from compromising other device drivers or applications.

Loki [36] shows how device drivers, among other parts of kernel code, can be made untrusted by using physical memory tagging. However, Loki incurs a memory overhead for storing tags, and requires modifying both the CPU and the DMA engines to perform tag checks, which is unlikely to appear in mainstream systems in the near future. Unlike SUD, Loki's memory tagging also cannot protect against devices issuing arbitrary interrupts.

Many of the in-kernel isolation techniques, including Nooks, SFI, and MMP, allow restarting a crashed device driver. However, doing so requires being able to reclaim all resources allocated to that driver at runtime, such as kernel memory, threads, stacks, and so on. By running the entire driver in an untrusted user-level process, SUD avoids this problem altogether.

Another approach to confining operations made by the hardware device on behalf of the driver is to rely on a declarative specification of the hardware's state machine, such as in Nexus [33] or Termite [28]. These techniques can provide fine-grained safety properties specific to each device, using a software reference monitor to control a driver's interactions with a device. However, if a specification is not available, or is incorrect, such a system would not be able to confine a malicious device driver, since it is impossible to predict how interactions between the driver and the device translate into DMA accesses initiated by the device.

In contrast to a specification-based approach, SUD enforces a single safety specification, namely, memory safety for PCI express devices. It does so without relying on precise knowledge of when a device might issue DMA requests or interrupts, by using hardware to control device DMA and interrupts, and providing additional system calls to allow driver manipulation of PCI register state. The drawback of enforcing a single memory safety property is that SUD cannot protect physical devices from corruption by misbehaving drivers, unlike [33]. We expect that the two techniques could be combined, by enforcing a base memory safety property in SUD, and using finer-grained specifications to ensure higher-level properties.

Nooks introduced the concept of shadow drivers [29] to recover device driver state after a fault, and SUD's architecture could also use shadow drivers to gracefully restart untrusted device drivers. In SUD, shadow drivers could execute either in fully-trusted kernel code, or in a separate untrusted user-space process, isolated from the untrusted driver they are shadowing. Techniques from CuriOS [8] could likewise be applied to address this problem.

Device driver reuse is another important area of related work. Some of the approaches to this problem have been to run device drivers in a virtual machine [22] without security guarantees, or to synthesize device drivers from a common specification language [28]. By allowing untrusted device drivers to execute in user-space, SUD simplifies the task of reusing existing, unmodified device drivers safely across different kernels. A well-defined driver interface, such as [28], would make it easier to move drivers to user-space, but SUD's architecture would still provide isolation.

Even if a driver cannot crash the rest of the system, it may fail to function correctly. A number of systems have been developed to help programmers catch common programming mistakes in driver code [19, 27], to make sure that the driver cannot mis-configure the physical device [33], and to guarantee that the driver implements the hardware device's state machine correctly [28]. A well-meaning driver running under SUD would benefit from all of these techniques, but the correctness of these techniques, or whether they were used at all, would not impact the isolation guarantees made by SUD.

Finally, user-space device drivers [21] provide a number of well-known advantages over running drivers in the kernel, including ease of debugging, driver upgrades, driver reuse, and fault isolation from many of the bugs in the driver code. Microdrivers [12] shows that the performance-critical aspects of a driver can be moved into trusted kernel code, while running the bulk of the driver code in user-space with only a small performance penalty, even if written in a type-safe language like Java [26].

SUD achieves the same benefits of running drivers in user-space, but does not rely on any device-specific trusted kernel code. This allows SUD to run arbitrary device drivers and applications with direct hardware access, without having to trust any part of them ahead of time, at the cost of somewhat higher CPU overheads as compared to Microdrivers. We expect that performance techniques from other user-level device driver systems [21] can be applied to SUD to similarly reduce the CPU overhead.

3 DESIGN

The goal of SUD is to prevent a misbehaving device driver from corrupting or disabling the rest of the system, including the kernel, applications, and other drivers.[2] At the same time, SUD strives to provide good performance in the common case of well-behaved drivers. SUD assumes that the driver can issue arbitrary instructions or system calls, and can also configure the physical device to issue arbitrary DMA operations or interrupts. The driver can also refuse to respond to any requests, or simply go into

[2]Of course, if the application relies on the device in question, such as a web server relying on the network card, the application will not be able to make forward progress until a working device driver is available.

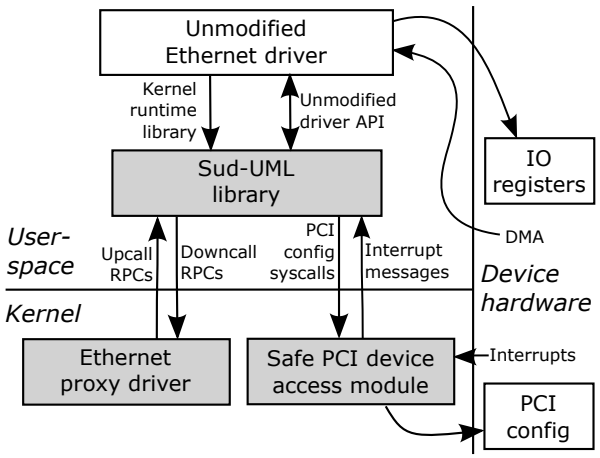

Figure 1: Overview of the interactions between components of SUD (shaded). Shown in user-space is an unmodified Ethernet device driver running on top of SUD-UML. A separate driver process runs for each device driver. Shown in kernel-space are two SUD kernel modules, an Ethernet proxy driver (used by all Ethernet device drivers in SUD), and a safe PCI device access module (used by all PCI card drivers in SUD). Arrows indicate request flow.

an infinite loop. To confine drivers, SUD assumes the use of recent *x*86 hardware, as we detail in Section 3.2.

The design of SUD consists of three distinct components, as illustrated in Figure 1. First, a *proxy driver* Linux kernel module allows user-space device drivers to implement the device driver interface expected by the Linux kernel. This kernel module acts as a proxy driver that can implement a particular type of device, such as an Ethernet interface or a wireless card. The proxy driver's job is to translate kernel calls to the proxy driver into upcalls to the user-space driver. A *safe PCI device access* kernel module allows user-space drivers to manage a physical hardware device, while ensuring that the driver cannot use the device to corrupt the rest of the system. Finally, a user-space library based on User-Mode Linux (UML) [9], called SUD-UML, allows unmodified Linux device drivers to run in untrusted user-space processes.

The rest of this section describes how the user-space device driver interacts with the rest of the system, focusing on how isolation is ensured for malicious device drivers.

3.1 API between kernel and driver

Traditional in-kernel device drivers interact with the kernel through well-known APIs. In Linux a driver typically registers itself with the kernel by calling a register function and passing a `struct` initialized with driver specific data and callbacks. The kernel invokes the callbacks to pass data and control to the driver. Likewise, drivers deliver data and execute kernel functions by calling predefined kernel functions.

As a concrete example, consider the kernel device driver for an imaginary "nic" Ethernet device, whose pseudo-code is shown in Figure 2. The Linux PCI

```
void nic_xmit_frame(struct sk_buff *skb)
{
    /*
     * Transmit a packet on behalf of the networking
     * stack.
     */
    nic_tx_buffer(skb->data, skb->data_len);
}

void nic_do_ioctl(int ioctl, char *result)
{
    /* Return MII media status. */
    if (ioctl == SIOCGMIIREG)
        nic_read_mii(result);
}

void nic_irq_handler(void)
{
    /*
     * Pass a recently received packet up to the
     * networking stack.
     */
    struct sk_buff *skb = nic_rx_skb();
    netif_rx(skb);
}

void nic_open(void)
{
    /*
     * Register an IRQ handler with the kernel and
     * enable the nic.
     */
    request_irq(nic_irq_num(),
                nic_irq_handler);
    nic_enable();
}

void nic_close(void)
{
    free_irq(nic_irq_num());
}

struct net_device_ops nic_netdev_ops = {
    .ndo_open       = nic_open,
    .ndo_stop       = nic_close,
    .ndo_start_xmit = nic_xmit_frame,
    .ndo_do_ioctl   = nic_do_ioctl,
};

int nic_probe(void)
{
    /* Register a device driver with the kernel. */
    char mac[6];
    struct net_device *netdev = alloc_etherdev();
    netdev->netdev_ops = &nic_netdev_ops;
    nic_read_mac_addr(mac);
    memcpy(netdev->dev_addr, mac, 6);
    register_netdev(netdev);
    return 0;
}
```

Figure 2: Example in-kernel Ethernet driver code, with five callback functions `nic_open`, `nic_close`, `nic_xmit_frame`, `nic_do_ioctl`, and `nic_irq_handler`. Ethernet drivers for real device require more lines of code.

code calls `nic_probe` when it notices a PCI device that matches the nic's PCI device and vendor ID. `nic_probe` allocates and initializes a `struct net_device` with the nic's MAC address and set of device specific callback functions then registers with Linux by calling `register_netdev`. When a user activates the device (*e.g.* by calling `ifconfig eth0 up`), Linux invokes the `nic_open` callback, which registers an IRQ handler to handle the device's interrupts, and enables the nic. When the Linux networking stack needs to send a packet, it passes the packet to the `nic_xmit_frame` callback. Likewise, when the nic receives a packet it passes the packet to the networking stack by calling `netif_rx`.

In order to move device drivers into user-space processes, SUD must translate the API between the kernel and the device driver, such as the example shown in Figure 2, into a message-based protocol that is more suitable for user-kernel communication. SUD uses proxy drivers for this purpose. A SUD proxy driver registers with the Linux device driver subsystem, providing any callback functions and data required by the class of drivers it supports. When the kernel invokes a proxy driver's callback function, the proxy driver translates the callback into an RPC call into a user-space driver. The kernel-driver API can also include shared memory that is accessed by either the driver or the kernel without invoking each other's functions. In our Ethernet driver example, the card's MAC address is stored in `netdev->dev_addr`, and is accessed without the use of any function calls. The proxy driver synchronizes such shared memory locations by mirroring, as we will discuss in Section 3.3.

SUD proxy drivers use a remote procedure call abstraction called *user channels* (or uchans for short), which we've optimized for messaging using memory shared between kernel and user address spaces. Figure 3 provides an overview of the SUD uchan interface. SUD implements uchans as special file descriptors. A uchan library translates the API in Figure 3 to operations on the file descriptor.

When the kernel invokes one of the proxy driver's callbacks, such as the function for transmitting a packet, the proxy driver marshals that request into an *upcall* into the user-space process. In the case of transmitting a packet, the proxy driver copies packet information into a `msg_t`, and calls `sud_asend` to add the `msg_t` to the queue holding kernel-to-user messages. Since transmitting a packet does not require an immediate reply, the proxy asynchronously sends the `msg_t`. On the other hand, synchronous upcalls are used for operations that require an immediate reply, such as `ioctl` calls to query the current MII media status of an Ethernet card, and result in the message being sent with `sud_send`, which blocks the callback until the user-space driver replies to the message.

kernel and user-space functions

`sud_send(msg_t)`	Send a synchronous message.
`sud_asend(msg_t)`	Send an asynchronous message.
`buf_t sud_alloc()`	Allocate a shared buffer.
`sud_free(buf_t)`	Free a shared buffer.

user-space functions

`msg_t sud_wait()`	Wait for a message.
`sud_reply(msg_t)`	Reply to a message.

Figure 3: Overview of the SUD uchan and memory allocation API.

The user-space process is responsible for handling kernel upcall messages, and typically the driver waits for a message from the kernel by calling `sud_wait`. When the proxy driver places on a message on the kernel-to-user queue, `sud_wait` dequeues the message and returns it to the user-space driver. The user-space driver processes the message by unmarshaling the arguments from the message, and invoking the corresponding callback in the driver code. If the callback returns a result (*i.e.* the kernel called `sud_send`), the user-space driver marshals the response into a `msg_t`, and calls `sud_reply`, which places the `msg_t` on a queue holding user-to-kernel messages.

When the user-to-kernel message queue contains a reply message, the proxy driver removes the message from the queue and unblocks the callback waiting for the reply. The callback completes by returning appropriate results to its caller. In the `ioctl` example, the kernel passes a buffer to the callback that the callback copies the result from the user-space driver reply into.

User-space drivers may also need to invoke certain kernel functions, such as changing the link status of the Ethernet interface. This is called a *downcall* in SUD, and is implemented in an analogous fashion, where the user-space driver and the in-kernel proxy driver reverse roles in the RPC protocol. One notable difference is that the kernel returns results of downcalls directly by copying the results into the message buffer the driver passed to `sud_send`, instead of by sending a separate message to the driver process.

3.1.1 Protecting the kernel from the device driver

Moving device drivers to user-space prevents device drivers from accessing kernel memory directly. This prevents buggy or malicious device drivers from crashing the kernel. However, a buggy or malicious user-space device driver may still break the kernel, other processes, or other devices, unless special precautions are taken at the user-kernel API. The kernel, and the proxy driver in particular, needs to make as few assumptions as possible about the behavior of the user-space device driver. The rest of this subsection describes how SUD handles liveness and semantic assumptions.

Liveness assumptions. One assumption that is often made of trusted in-kernel drivers is that they will handle

requests in a timely fashion. However, if a malicious user-space device driver fails to respond to upcalls, many threads in the kernel may eventually be blocked waiting for responses that will never arrive. SUD addresses this problem in two ways. First, for upcalls that require a response from the user-space device driver before the in-kernel proxy can proceed, the upcall is made *interruptable*. This allows the user to abort (`Ctrl-C`) an operation that appears to have hung, such as running `ifconfig` on an unresponsive driver. To implement interruptable upcalls, the user-kernel interface must be carefully designed to allow any synchronous upcall to return an error.

Second, SUD uses *asynchronous* upcalls whenever possible. Asynchronous upcalls can be used in situations where the in-kernel proxy driver does not require any response from the user-space driver in order to return to its caller safely, such as packet transmission. If the device driver's queue is full, the kernel can wait a short period of time to determine if the user-space driver is making any progress at all, and if not, the driver can be reported as hung to the user.

Asynchronous upcalls are also necessary for handling upcalls from threads running in a non-preemptable context, such as when holding a spinlock. A thread in a non-preemptable context cannot go to sleep, and therefore cannot allow the user-space driver to execute and process the upcall. While multi-core systems can avoid this problem by running the driver and the non-preemptable context concurrently, SUD still must not rely on the liveness of the untrusted device driver.

A potential problem can occur if a non-preemptable kernel thread invokes the in-kernel proxy driver and expects a response (so that the proxy driver might need to perform an upcall). One solution to this problem is rewriting the kernel code so a non-preemptable context is unnecessary. In Linux, for example, we could replace the spin lock with a mutex. However, this solution is undesirable, because it might require restructuring portions of the kernel and affect performance poorly.

To address this problem, we observe that the work performed by functions called in a non-preemptable context is usually small and well-defined; after all, the kernel tries to avoid doing large amounts of work in a non-preemptable context. Thus, for every class of devices, the corresponding SUD proxy driver implements any short functions invoked by the kernel as part of the driver API.[3] Any state required by these functions is mirrored and synchronized between the real kernel and the SUD-UML kernel. For example, the Linux 802.11 network stack calls the driver to enable certain features, while executing in a non-preemptable context; the driver must respond with the features it supports and will enable. The wireless proxy driver mirrors the (static) supported feature set, and when the kernel invokes the function to enable some feature, the proxy driver queues an asynchronous upcall to SUD-UML containing the newly-enabled features.

Semantic assumptions. A second class of assumptions that kernel code may make about trusted in-kernel drivers has to do with the *semantics* of the driver's responses. A hypothetical kernel might rely on the fact that, once the kernel changes the MAC address of an Ethernet card, a query to get the current MAC address will return the new value. SUD does not enforce such invariants on drivers, because we have not found any examples of such assumptions in practice. Infact, Linux subsystems that interact with device drivers (such as the network device subsystem) are often robust to driver mistakes, and print error messages when the driver is acting in unexpected ways. At this point, the administrator can kill the misbehaving user-space driver. If the kernel did rely on higher-level invariants about driver behavior, the corresponding proxy driver would need to be modified to keep track of the necessary state, and to enforce the invariant.

3.1.2 Uchan optimizations

Two potential sources of performance overhead in SUD come from the context switches due to upcalls and downcalls, and from data copying, such as the packets in an Ethernet driver.

The SUD uchan implementation optimizes the number of context switches due to upcalls and downcalls by mapping message queues into memory shared by the kernel and user-space driver. SUD uchans implement the message queues using ring buffers. The kernel writes messages into the head of the kernel-to-user ring. When the user-space driver calls `sud_wait` to wait for a message, `sud_wait` polls the kernel-to-user ring tail pointer. If the tail points to a message `sud_wait` dequeues the message by incrementing the tail pointer, the user-space driver processes the message, and possibly returns results by calling `sud_reply`. When the tail of the ring equals the head, the queue is empty and the user-space process sleeps by calling `select` on the uchan file descriptor. `select` returns when the kernel adds a message to the head of the kernel-to-user ring. This interface allows a user-space process to process multiple messages without entering the kernel.

The downcall message queues work in a similar fashion, except that the user-space driver writes to the head of the ring and the kernel reads from the tail of the ring. When a user-space driver calls `sud_asend`, the uchan library adds the message to the queue, but does not notify the kernel

[3] While we have found that this approach works for device drivers we have considered so far, it is possible that other kernel APIs have long, device-specific functions invoked in a non-preemptable context. Supporting these devices in SUD would require modifying the kernel, as Section 3.1.3 discusses.

of the pending message until the user-space driver calls `sud_wait` or `sud_send`. This allows user-space drivers to batch asynchronous downcalls.

SUD also optimizes data copying overhead by pre-allocating data buffers in the user-space driver, and having the in-kernel proxy driver map them in the kernel's address space. A call to `sud_alloc` returns one of the shared messages buffers and `sud_free` returns the message buffer to the shared heap. In an Ethernet driver, this allows packet transmit upcalls and packet receive downcalls to exchange pointers using `sud_send`, and avoid copying the data. As we will describe later, the same shared buffers are passed to the physical device to access via DMA, avoiding any additional data copy operations in the user-space driver.

The in-kernel proxy driver may need to perform one data copy operation to guard against malicious user-space drivers changing the shared-memory data after it has been passed to the kernel. For example, a malicious driver may construct an incoming packet to initially look like a safe packet that passes through the firewall, but once the firewall approves the packet, the malicious driver changes the packet in shared memory to instead connect to a firewalled service. In the case of network drivers, we can avoid the overhead of this additional data copy operation by performing it at the same time that the packet's checksum is computed and verified, at which point the data is already being brought into the CPU's data cache. An alternative design may be to mark the page table entries read-only, but we have found that invalidating TLB entries from the IOMMU's page table is prohibitively expensive on current hardware.

3.1.3 Limitations

The implementation of the Linux kernel imposes several limitations on what types of device drivers SUD supports and what driver features a proxy driver can support.

It is unlikely SUD will ever be able to support device drivers that are critical for a kernel to function. For example, Linux relies on a functioning timer device driver to signal the scheduler when a time quantum elapses. A buggy or malicious timer driver could deadlock the kernel, even while running as a SUD user-space driver.

Another limitation is how proxy drivers handle callbacks when the calling kernel thread is non-preemptable. Servicing the callback in the in-kernel proxy allow SUD to support common functions for several device classes, but it does not work in all cases. For SUD to support all the functions of Linux kernel devices it is likely that some kernel subsystems would need to be restructured so non-preemptable threads do not need to make upcalls.

Despite these limitations SUD supports common features for several widely used devices. We could incrementally add support for more functions.

3.2 Confining hardware device access

The key challenge to isolating an untrusted device driver is making sure that a malicious driver cannot misuse its access to the underlying hardware device to escape isolation. In this subsection, we discuss how SUD confines the driver's interactions with the physical device, first focusing on operations that the driver can perform on the device, and second discussing the operations that the device itself may be able to perform.

To control access to devices without knowing the details of the specific device hardware interface, SUD assumes that all devices managed from user-space are PCI devices. This assumption holds for almost all devices of interest today.

3.2.1 Driver-initiated operations

In order for the user-space device driver to function, it must be able to perform certain operations on the hardware device. This includes accessing the device's memory-mapped IO registers, accessing legacy *x*86 IO registers on the device, and accessing the device's PCI configuration registers. SUD's safe PCI device access module, shown in Figure 1, is responsible for supporting these operations.

To allow access to memory-mapped IO registers, SUD's PCI device access module allows a user-space device driver to directly map them into the driver's address space. To make sure that these page mappings do not grant unintended privileges to an untrusted device driver, SUD makes sure that all memory-mapped IO ranges are page-aligned, and does not allow untrusted drivers to access pages that contain memory-mapped IO registers from multiple devices.

Certain devices and drivers also require the use of legacy *x*86 IO-space registers for initialization. To allow drivers to access the device's IO-space registers, SUD uses the IOPB bitmask in the task's TSS [16] to permit access to specific IO ports.

Finally, drivers need to access the PCI configuration space of their device to set certain PCI-specific parameters. However, some of the PCI configuration space parameters might allow a malicious driver to intercept writes to arbitrary physical addresses or IO ports, or issue PCI transactions on behalf of other devices. To prevent such attacks, SUD exposes PCI configuration space access through a special system call interface, instead of granting direct hardware access to the user-space driver. This allows SUD to ensure that sensitive PCI configuration registers are not modified by the untrusted driver.

3.2.2 Device-initiated operations

A malicious user-space driver may be able to escape isolation by asking the physical hardware device to perform

operations on its behalf, such as reading or writing physical memory via DMA, or raising arbitrary interrupts. To prevent such problems, SUD uses hardware mechanisms, as shown in Figure 4, to confine each physical device managed by an untrusted device driver.

DMA. First and foremost, SUD must ensure that the device does not access arbitrary physical memory via DMA. To do so, SUD relies on IOMMU hardware available in recent Intel [17] and AMD [3] CPUs and chipsets to interpose on all DMA operations. The PCI device access module specifies a page table for each PCI device in the system, and the IOMMU translates the addresses in each DMA request according to the page table for the originating PCI device, much like the CPU's MMU. By only inserting page mappings for physical pages accessible to the untrusted driver into the corresponding PCI device's IO page table, SUD ensures that a PCI device cannot access any physical memory not already accessible to the untrusted driver itself.

Peer-to-peer DMA. Although an IOMMU protects the physical memory of the system from device DMA requests, a subtle problem remains. Traditional PCI bridges route DMA transactions according to the destination physical address, and a PCI device under the control of a malicious driver may be able to DMA into the memory-mapped registers of another PCI device managed by a different driver. As can be seen in Figure 4, a DMA transaction from device A to the physical address of device B's registers would not cross the IOMMU, and thereby would not be prevented.

To avoid this problem, SUD requires the use of a PCI express bus, which uses point-to-point physical links between PCI devices and switches, as opposed to traditional PCI which uses a real bus shared by multiple devices. When multiple devices share the same physical PCI bus, there is nothing that can prevent a device-to-device DMA attack. With PCI express, at least one PCI express switch is present between any two devices, and can help us avoid this problem.

To ensure that all PCI requests pass through the root switch, SUD enables PCI access control services (ACS) [25] on all PCI express switches. ACS allows the operating system to control the routing and filtering of certain PCI requests. In particular, SUD enables source validation, which ensures that a downstream PCI device cannot spoof its source address, and P2P request and completion redirection, which ensures that all DMA requests and responses are always propagated from devices to the root (where the IOMMU is located), and from the root to the devices, but never from one device to another.

Interrupts. The final issue that SUD must address is interrupts that can be raised by devices. Although interrupts are unlikely to corrupt any system state on their own, a malicious driver could use an interrupt storm to force CPUs to keep handling interrupts, thereby livelocking the system. Traditionally, the device driver's interrupt handler is responsible for clearing the interrupt condition before interrupts are re-enabled. In some cases devices share an interrupt and are expected to coordinate interrupt handling. With untrusted device drivers, however, the kernel cannot rely on a driver to clear the interrupt condition or cooperate with other drivers, so SUD takes a few measures to prevent this from happening.

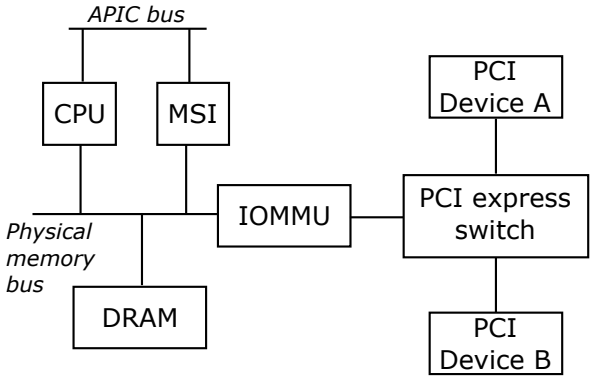

Figure 4: Overview of the hardware mechanisms used by SUD to confine hardware devices managed by untrusted drivers. In some systems, the APIC bus is overlaid on the physical memory bus, and in some systems the DRAM is attached directly to the CPU.

First, SUD prevents devices from raising legacy interrupts shared by multiple devices. SUD does so by restricting drivers from directly accessing the PCI configuration registers to change the interrupt configuration. Instead of legacy interrupts, SUD relies on message-signaled interrupts (MSI), which are mandatory for PCI express devices, and support generic interrupt masking that does not depend on the specific device.

Second, SUD forwards device interrupts to untrusted drivers via the upcall mechanism that was described in Section 3.1. When an interrupt comes in, SUD issues an upcall to the corresponding driver indicating that an interrupt was signaled. At this point, SUD does not mask further MSI interrupts, since they are edge-triggered. However, if another interrupt for the same device comes in, before the driver indicates that it has finished processing the interrupt, SUD uses MSI to make sure further interrupts do not prevent the driver's forward progress.

Note that this design allows the OS scheduler to schedule device drivers along with other processes in the system. When the device driver's time quantum expires, it will be descheduled, and even if it were handling an interrupt, MSI will be used to mask subsequent interrupts until the driver can run again. Of course, in practice it may be desirable to run device drivers with high priority, to make sure that devices are serviced promptly, but should

a driver misbehave, other processes will still be able to execute.

The final consideration has to do with how message-signaled interrupts are implemented on *x*86. A device signals an MSI by performing a memory write to a reserved physical memory address, and the MSI controller on the other side of the IOMMU picks up writes to that memory address and translates them into CPU interrupts on the APIC bus, as shown in Figure 4. Unfortunately, it is impossible to determine whether a write to the MSI address was caused by a real interrupt, or a stray DMA write to the same address. SUD can mask real interrupts by changing the MSI register in the PCI configuration space of the device, but cannot prevent stray DMA writes to the MSI address.

To avoid interrupt storms and arbitrary interrupts caused by a malicious driver using DMA to the MSI address, SUD uses two strategies, depending on which IOMMU hardware it has available to it. On Intel's VT-d [17], SUD uses interrupt remapping support. Interrupt remapping allows the OS kernel to supply a table translating potential MSI interrupts raised by each device to the physical interrupt that should be raised as a result, if any. Changing an interrupt remapping table is more expensive than using MSI masking, so SUD first tries to use MSI to mask an interrupt, and if that fails, SUD changes the interrupt remapping table to disable MSI interrupts from that device altogether. With AMD's IOMMU [3], SUD removes the mapping for the MSI address from that particular device's IO page table, thereby preventing that device from performing any MSI writes.

3.3 Running unmodified drivers in user-space

To allow unmodified Linux device drivers to run in untrusted user-space processes, SUD uses UML [9] to supply a kernel-like runtime environment in user-space, which we call SUD-UML. SUD-UML's UML environment provides unmodified drivers with all of the kernel code they rely on, such as `kmalloc()` and `jiffies`.

SUD-UML differs from traditional UML in three key areas that allow it to connect drivers to the rest of the world. First, SUD-UML replaces low-level routines that access PCI devices and allocate DMA memory with calls to the underlying kernel's safe PCI device access module, shown in Figure 1. For example, when the user-space driver calls `pci_enable_device` to enable a PCI device, SUD-UML performs a downcall to the underlying kernel to do so. When the user-space driver allocates DMA-capable memory, SUD-UML requests that the newly-allocated memory be added to the IOMMU page table for the relevant device. When the driver registers an interrupt handler, SUD-UML asks the underlying kernel to forward interrupt upcalls to it.

Second, SUD-UML implements the user-kernel RPC interface that we described in Section 3.1 by invoking the unmodified Linux driver when it receives an upcall from the kernel, and sends the response, if any, back to the kernel over the same file descriptor. For example, when an interrupt upcall is received by SUD-UML, it invokes the interrupt handler that was registered by the user-space driver.

Finally, SUD-UML mirrors shared-memory state that is part of the driver's kernel API, by maintaining the same state in both the real kernel and SUD-UML's UML kernel, and synchronizing the two copies as needed. For example, the Linux kernel uses shared memory variables to track the link state of an Ethernet interface, or the currently available bitrates for a wireless card. The SUD proxy driver and SUD-UML cooperate to synchronize the two copies of the state. If the proxy driver updates the kernel's copy of the state, it sends an upcall to SUD-UML with the new value. In SUD-UML, we exploit the fact that updates to driver shared memory variables are done via macros, and modify these macros to send a downcall with the new state to the real kernel. This allows the user-space device driver to remain unchanged.

Updates to shared-memory state are ordered with respect to all other upcall and downcall messages, which avoids race conditions. Typically, any given shared-memory variable is updated by either the device driver or by the kernel, but not both. As a result, changes to shared-memory state appear in the correct order with respect to other calls to or from the device driver. However, for security purposes, the only state that matters is the state in the real kernel. As discussed in Section 3.1.1, the Linux kernel is robust with respect to semantic assumptions about values reported by device drivers.

Poorly written or legacy drivers often fail to follow kernel conventions for using system resources. For example, some graphics cards set up DMA descriptors with physical addresses instead of using the kernel DMA interface to get a DMA capable address. A poorly written driver will run in SUD-UML until it attempts to accesses a resource that it has not properly allocated. When this happens the SUD-UML process terminates with an error.

4 IMPLEMENTATION

We have implemented a prototype of SUD as a kernel module for Ubuntu's Linux 2.6.30-5 kernel on x86-64 CPUs. We made several minor modifications to the Linux kernel proper. In particular, we augmented the DMA mapping interface to include functions for flushing the IOTLB, mapping memory starting at a specified IO virtual address, and garbage collecting an IO address space. We have only tested SUD on Intel hardware with VT-d [17], but the implementation does not rely on VT-d features, and SUD should run on any IOMMU hardware that provides DMA

Feature	Lines of code
Safe PCI device access module	2800
Ethernet proxy driver	300
Wireless proxy driver	600
Audio card proxy driver	550
USB host proxy driver	0
SUD-UML runtime	5000

Figure 5: Lines of code required to implement our prototype of SUD, SUD-UML, and each of the device-specific proxy drivers. SUD-UML uses about 3 Mbytes of RAM, not including UML kernel text, for each driver process.

address translation, such as AMD's IOMMU [3]. SUD-UML, our modified version of UML, is also based on Ubuntu's Linux 2.6.30-5 kernel.

Our current prototypes of SUD and SUD-UML include proxy drivers and UML support for Ethernet cards, wireless cards, sound cards, and USB host controllers and devices. On top of this, we have been able to run a range of device drivers as untrusted user-space processes, including the e1000e Gigabit Ethernet card driver, the iwlagn5000 802.11 wireless card driver, the ne2k-pci Ethernet card driver, the snd_hda_intel sound card driver, the EHCI and UHCI USB host controller drivers, and various USB device drivers, all with no modifications to the driver itself.

Figure 5 summarizes the lines of code necessary to implement the base SUD system, the base SUD-UML environment, and to add each class of devices. The USB host driver class requires no code beyond what is provided by the SUD core. Some USB devices, however, require additional driver classes. For example, a USB 802.11 wireless adapter can use the existing wireless proxy driver, and a USB sound card could use the audio card proxy driver. We are working on a block device proxy driver to support USB storage devices.

4.1 User-mode driver API

SUD's kernel module exports four SUD device files for each PCI device that it manages, as shown in Figure 6. The device files are initially owned by root, but the system administrator can set the owner of these devices to any other UID, and then run an untrusted device driver for this device under that UID.

When the system administrator starts a driver, SUD-UML searches `sysfs` for a matching device. If SUD-UML finds a matching device, it invokes the kernel to start a proxy driver and open a uchan shared with the proxy driver.

SUD-UML translates Linux kernel device driver API calls to operations on the SUD device files. When a device driver calls `dma_alloc_coherent`, SUD-UML uses `mmap` to allocate anonymous memory from the `dma_coherent` device. This allocates the requested number of memory pages in the driver's process, and also maps the same pages at the same virtual address in the corresponding device's IOMMU page table. Likewise, SUD-UML allocates cacheable DMA memory using anonymous `mmap` on `dma_caching`. The `mmio` file exports the PCI device's memory-mapped IO registers, which the driver accesses by `mmap`ing this device. Finally, The `ctl` file is used to handle kernel upcalls and to issue downcalls. Figure 7 gives sample of upcalls and downcalls.

System administrators can manage user-space drivers in the same way they manage other processes and daemons. An administrator can terminate a misbehaving or buggy driver with `kill -9`, and restart it by starting a new SUD-UML process for the device. The Linux functions for managing resource limits, such as `setrlimit`, work for user-space drivers. For example, an administrator might use `setrlimit` to limit the amount of memory a suspicious driver is allowed to allocate.

Some device drivers, such as sound card drivers, might require real time scheduling constraints to function properly, for example to ensure a high bit rate. For these devices an administrator can use `sched_setscheduler` to assign the user-space driver one of Linux' real-time scheduling policies. If the audio driver turns out to be malicious or buggy, it could consume a large fraction of CPU time, but unlike a fully-trusted kernel driver, it would not be able to lock up the system entirely.

4.2 Performance optimizations

Most of the performance overhead in SUD comes from SUD-UML. Our efforts to optimize SUD-UML are ongoing, but we have implemented several important optimizations, which we describe in the following paragraphs.

One optimization is to handle upcalls and invoke callbacks directly from the UML idle thread. This must be done with care, however, because some drivers implement callbacks that block the calling thread, but expect other threads to continue to invoke driver callbacks. For example, the e1000e driver determines which type of interrupt to configure (*e.g.* legacy or MSI) by triggering the interrupt, sleeping, and then checking a bit that should have been set by the e1000e interrupt handler.

To handle this case, when the UML idle thread receives an upcall, it checks if the corresponding callback is allowed to block (according to kernel conventions). If the callback is not allowed to block, the UML idle thread invokes the callback directly. Otherwise, the idle thread creates and runs a worker thread to invoke the callback. We optimize worker thread creation using a thread pool.

Another optimization, which we have not implemented yet, but we expect to improve performance, it to use superpages to map SUD-UML's memory, including memory shared with the kernel. The kernel must flush all non-kernel mappings when it performs a context switch between user-space virtual address spaces. This impacts

File	Use
/sys/devices/.../sud/ctl	Transfers upcall and downcall messages.
/sys/devices/.../sud/mmio	Represents the PCI device's memory-mapped IO regions; intended for use with mmap.
/sys/devices/.../sud/dma_coherent	Allocates anonymous non-caching memory on mmap, mapped at the same virtual address in both the driver's page table, and the device's IOMMU page table.
/sys/devices/.../sud/dma_caching	Allocates anonymous caching memory on mmap, mapped at the same virtual address in both the driver's page table, and the device's IOMMU page table.

Figure 6: An overview of device files that SUD exports for each PCI device.

Upcall	Description
ioctl	Request that the driver perform a device-specific ioctl.
interrupt	Invoke the SUD-UML driver interrupt handler.
net_open	Prepare a network device for operation.
bss_change	Notify an 802.11 device that the BSS has changed.

Downcall	Description
interrupt_ack	Request that SUD unmask the device interrupt line.
request_region	Add IO-space ports to the driver's IO permission bitmask.
netif_rx	Submit a received packet to the kernel's network stack.
pci_find_capability	Checks if device supports a particular capability.

Figure 7: A sample of SUD upcalls and downcalls.

user-space drivers, because the drivers often have a large working set of DMA buffers. For example, the e1000e allocates 256 buffers, which might span multiple pages, for both the transmit and receive DMA rings. In the case of the e1000e driver, the driver might read the contents of the DMA buffer after a packet has been received. This results in a TLB miss if the kernel context-switched from the SUD-UML process to another process since the last time the driver read the DMA buffer. By mapping all the DMA buffers using several super pages, SUD-UML could avoid many of these TLB misses.

5 EVALUATION

To evaluate SUD, we wanted to understand how hard it is to use SUD, how well it performs at runtime, and how well it protects the system from malicious drivers. The implementation section already illustrated that SUD allows existing Linux device drivers to be used as untrusted user-space drivers with no source-code modifications. In this section, we focus on the runtime overheads imposed by SUD for running device drivers in user-space, and on the isolation guarantees that SUD provides.

In short, our results show that SUD incurs performance overheads comparable to other device driver isolation techniques, while providing stronger guarantees. To illustrate SUD's security guarantees, we have verified that SUD prevents both DMA and interrupt attacks, as well as the driver's attempts to mishandle kernel upcalls. The rest of this section describes our experimental evaluation in more detail.

5.1 Network driver performance

The primary performance concern in running device drivers as user-space processes is the overhead of context switching and copying data. To understand the performance overhead imposed by SUD, we consider the *worst-case* scenario—a Gigabit Ethernet device that requires both high throughput and low latency to achieve high performance, using both small and large packets—so that any overhead introduced by SUD's protection mechanisms will show up clearly in the benchmark results. In practice, we expect most of the drivers running under SUD to be less performance-critical (for example keyboard, mouse, printer, or sound card drivers), and thus any performance penalties imposed by SUD on those drivers would be even less noticeable.

We run four netperf [18] benchmarks to exercise the e1000e Linux device driver running under SUD on an Thinkpad X301 with a 1.4GHz dual-core Intel Centrino. The Thinkpad is connected to a 2.8GHz dual-core Pentium D Dell Optiplex by a Gigabit switched network. We configure netperf to run experiments to report results accurate to 5% with 99% confidence.

Figure 8 summarizes performance results and CPU overheads for the untrusted driver running in SUD and the trusted driver running in the kernel. The first benchmark, TCP_STREAM, measures TCP receive throughput and is run with 87380 byte receive buffers and 16384 byte send buffers. SUD offers the same performance as the kernel driver with little overhead, because SUD-UML is able to batch delivery of many large packets to the kernel in one downcall.

The UDP_STREAM TX and RX benchmarks measure throughput for transmitting and receiving 64 byte UDP packets. These benchmarks are more CPU intensive than TCP_STREAM, and demonstrate overheads in SUD that might have been obscured by the use of large packets in TCP_STREAM. For both TX and RX, SUD performs comparably to the kernel driver, but has about a 11% overhead for TX and a 30% overhead for RX.

The final benchmark, UDP_RR, is designed to measure driver latency. The UDP_RR results are given in transac-

tions per second. The client completes a transaction when it sends a 64 byte UDP packet and waits for the server to reply with a 64 byte UDP packet. In some ways this is a worst case benchmark for SUD, which has a CPU overhead of 2x. After each packet transmit or receive the e1000e process sleeps to wait for the next event. Unfortunately, waking up the sleeping process can take as long as $4\mu s$ in Linux. If e1000e driver has more work and sleeps less, such as in the `UDP_STREAM` benchmark, this high wakeup overhead is avoided.

5.2 Security

We argued in Section 3 that SUD uses IOMMUs to prevent devices from accessing arbitrary physical memory. Figure 9 shows the IO virtual memory mappings for the e1000e driver. We read all mappings by walking the e1000e device's IO page directory. This ensures that the BIOS or other system software does not create special mappings for device use. The lack of any other mappings indicates that a malicious device driver can at most corrupt its own transmit and receive buffers, or raise an interrupt using MSI.

Our experimental machine does not have support for interrupt remapping in its IOMMU hardware, so our configuration is vulnerable to livelock by a malicious driver issuing DMA requests to the MSI address. Unfortunately, Intel VT-d always includes an implicit identity mapping for the MSI address in every page table, so it was not possible to prevent this type of attack. A newer chipset version would have avoided this weakness, and we expect that doing so would not impact the performance of SUD. Alternatively, AMD's IOMMU does not include an implicit MSI mapping, and we could simply unmap the MSI address on an AMD system when an interrupt storm is detected, to prevent further interrupts from a device.

We tested SUD's security by constructing explicit test cases for the attacks we described earlier in Section 3, including arbitrary DMA memory accesses from the device and interrupt storms. In all cases, SUD safely confined the device and the driver. We have also relied on SUD's security guarantees while developing SUD-UML and testing drivers. For example, in one situation a bug in our SUD-UML DMA code was returning an incorrect DMA address, which caused the USB host controller driver to attempt a DMA to an unmapped address. The bug, however, was easy to spot, because it triggered a page fault. As another example, the SUD-UML interrupt code responsible for handling upcalls was not invoking the iwlagn5000 interrupt handler, but was re-enabling interrupts with SUD. The resulting interrupt storm was easily fixed by killing the SUD-UML process. It is also relatively simple to restart a crashed device driver by restarting the device driver process.

Test	Driver	Throughput	CPU %
`TCP_STREAM`	Kernel driver	941 Mbits/sec	12%
	Untrusted driver	941 Mbits/sec	13%
`UDP_STREAM` TX	Kernel driver	317 Kpackets/sec	35%
	Untrusted driver	308 Kpackets/sec	39%
`UDP_STREAM` RX	Kernel driver	238 Kpackets/sec	20%
	Untrusted driver	235 Kpackets/sec	26%
`UDP_RR`	Kernel driver	9590 Tx/sec	5%
	Untrusted driver	9489 Tx/sec	10%

Figure 8: TCP streaming, minimum-size UDP packet streaming, and UDP request-response performance for the e1000e Ethernet driver running as an in-kernel driver and as an untrusted SUD driver. Each UDP packet is 64-bytes.

Memory use	Start	End
TX ring descriptor	0x42430000	0x42431000
RX ring descriptor	0x42431000	0x42433000
TX buffers	0x42433000	0x42C33000
RX buffers	0x42C33000	0x43433000
Implicit MSI mapping	0xFEE00000	0xFEF00000

Figure 9: The IO virtual memory mappings for the e1000e driver.

6 DISCUSSION

We think SUD demonstrates that unmodified device drivers can be run as user-space processes with good performance. This section examines some limitations of SUD and explores directions of future work.

New hardware. Our test machine does not support interrupt remapping, which leaves SUD vulnerable to a livelock attack from a malicious driver. The ability to remap interrupts is necessary to prevent this attack, but could also be useful for improving performance. For example, it might be faster to mask an interrupt by remapping the MSI page instead of by reconfiguring the PCI device.

Hardware queued IOTLB invalidation, which is present in some Intel VT-d implementations, allows software to queue several IOTLB invalidations efficiently. SUD could use this feature to unmap DMA buffers from the user-space device driver while they are being processed by the kernel.

Device delegation. In the current SUD design, the kernel defines all of the devices in the system (e.g., all PCI devices), and grants user-space drivers access at that granularity (e.g., one PCI device). An alternative approach that we hope to explore in the future is to allow one untrusted device driver to create new device objects, which could then be delegated to separate device driver processes. For example, the system administrator might start a PCI express bus process, which would scan the PCI express bus and start a separate driver process for each device it found. If one of the devices on the PCI express bus was a USB host controller, the USB host controller driver might start a new driver process for each USB device it found. If the device was a SATA controller, the SATA driver may likewise start a new driver for each disk.

Finally, a network card with hardware support for multiple virtual queues, such as the Intel IXGBE, could give applications direct access to one of its queues.

Optimized drivers. Supporting unmodified device drivers is a primary goal SUD-UML. However, porting drivers to a SUD interface might eliminate some CPU overhead that results from supporting unmodified drivers. For example, SUD-UML constructs Linux socket buffers for each packet the kernel transmits, because this is what the unmodified device expects. By modifying device drivers to take advantage of the SUD interface directly, we may be able to achieve lower CPU overheads as in [21].

Applications. There are some applications that are not necessarily suitable to run in the kernel, but that benefit from direct access to hardware. These applications either make do with sub-optimal performance, or are implemented as trusted modules and run in the kernel, in spite of the security concerns. For example, the Click [20] router runs as a kernel module so that it has direct access to packets as they are received by the network card. With SUD, these applications could run as untrusted SUD-UML driver processes, with direct access to hardware, and achieve good performance without the security threat.

7 CONCLUSION

SUD is a new system for confining buggy or malicious Linux device drivers. SUD confines malicious drivers by running them in untrusted user-space processes. To ensure that hardware devices controlled by untrusted drivers do not compromise the rest of the system through DMA or interrupt attacks, SUD uses IOMMU hardware, PCI express switches, and message-signaled interrupts. SUD can run untrusted drivers for arbitrary PCI devices, without requiring any specialized language or specification. Our SUD prototype demonstrates support for Ethernet cards, wireless cards, sound cards, and USB host controllers, and achieves performance equal to in-kernel drivers with reasonable CPU overhead, while providing strong isolation from malicious drivers. SUD requires minimal changes to the Linux kernel—a total of two kernel modules comprising less than 4,000 lines of code—which may finally help these research ideas to be applied in practice.

ACKNOWLEDGMENTS

We thank Austin Clements and M. Frans Kaashoek for their feedback on this paper. Thanks to the anonymous reviewers, and to our shepherd, Jaeyeon Jung, for helping improve this paper as well.

REFERENCES

[1] Linux kernel i915 driver memory corruption vulnerability. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3831`.

[2] The L4Ka Project. `http://l4ka.org/`.

[3] Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification*, February 2006.

[4] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing IOMMUs for virtualization in linux and xen. In *Proceedings of the2006 Ottawa Linux Symposium*, Ottawa, Canada, July 2006.

[5] L. Butti and J. Tinnes. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology*, 4(1), Feburary 2008.

[6] M. Castro, M. Costa, J. P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[7] P. Chubb. Linux kernel infrastructure for user-level device drivers. In *In Linux Conference*, 2004.

[8] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.

[9] J. Dike. The user-mode Linux kernel home page. `http://user-mode-linux.sf.net/`.

[10] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[11] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proceedings of the 20th USENIX Large Installation System Administration Conference*, Washington, DC, December 2006.

[12] V. Ganapathy, M. Renzelmann, A. Balakrishnan, M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, March 2008.

[13] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[14] H. Härtig, J. Loeser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O architecture for microkernel-based operating systems. Technical Report TUD-FI03-08, TU Dresden, Dresden, Germany, July 2003.

[15] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum. Fault isolation for device drivers. In *Proceedings of the2009 IEEE Dependable Systems and Networks Conference*, Lisbon, Portugal, June–July 2009.

[16] Intel. *Intel 64 and IA-32 Architectures Developer's Manual*, November 2008.

[17] Intel. Intel Virtualization Technology for Directed I/O, September 2008.

[18] R. Jones. Netperf: A network performance benchmark, version 2.45, 2009. `http://www.netperf.org`.

[19] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[20] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(4):263–297, November 2000.

[21] B. Leslie, P. Chubb, N. Fitzroy-dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20, 2005.

[22] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[23] A. Menon, S. Schubert, and W. Zwaenepoel. TwinDrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest os drivers. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, March 2009.

[24] PCI-SIG. *PCI local bus specification*, revision 3.0 edition, February 2004.

[25] PCI-SIG. *PCI Express 2.0 base specification*, revision 0.9 edition, September 2006.

[26] M. J. Renzelmann and M. M. Swift. Decaf: Moving device drivers to a modern language. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, June 2009.

[27] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, March 2009.

[28] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[29] M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4), November 2006.

[30] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), November 2004.

[31] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 1997.

[32] VMware. Configuration examples and troubleshooting for VMDirectPath. `http://www.vmware.com/pdf/vsp_4_vmdirectpath_host.pdf`.

[33] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[34] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2002.

[35] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[36] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, pages 225–240, San Diego, CA, December 2008.