MoIO: Run-Time Monitoring for I/O Protocol Violations in Storage Device Drivers

Domenico Cotroneo, Luigi De Simone, Francesco Fucci, Roberto Natella
Università degli Studi di Napoli Federico II, Naples, Italy
{*cotroneo, luigi.desimone, francesco.fucci, roberto.natella*}*@unina.it*

***Abstract*—Bugs affecting storage device drivers include the so-called *protocol violation bugs*, which silently corrupt data and commands exchanged with I/O devices. Protocol violations are very difficult to prevent, since testing device driver is notoriously difficult. To address them, we present a monitoring approach for device drivers (MoIO) to *detect I/O protocol violations* at run-time. The approach *infers* a model of the interactions between the storage device driver, the OS kernel, and the hardware (the *device driver protocol*) by analyzing execution traces. The model is then used as a reference for detecting violations in production. The approach has been designed to have a low overhead and to overcome the lack of source code and protocol documentation. We show that the approach is feasible and effective by applying it on the SATA/AHCI storage device driver of the Linux kernel, and by performing fault injection and long-running tests.**

***Keywords*-Device Drivers; Storage failures; Run-time Monitoring; Model Inference; Linux kernel**

I. Introduction

It is well-known that device drivers are the most bug-prone part of the OS [1], [2], [3], and represent a critical component of every storage stack in IT systems. A large-scale study of failures from about 39,000 commercially-deployed storage systems [4], [5] showed that disk faults are not anymore the dominant factor of storage failures, and that software faults in the *I/O protocol stack* account for a noticeable percentage (up to 10%). These failures are caused by the so-called *protocol violation bugs*, i.e., bugs that violate the protocol between the hardware and the driver, such as misinterpreting or incorrectly setting the device state, and exchanging incorrect data and commands with the I/O device. Ryzhyk et al. [6] analyzed about 500 bugs in Linux device drivers, and found that a large proportion (38%) were indeed device protocol faults. Unfortunately, protocol violations are very difficult to prevent, since testing device drivers is notoriously difficult [7], [8]. Moreover, they have a severe impact, since data corruptions can spread without being noticed until they impact on end-users, when it is too late and the chances of data recovery are reduced.

In this paper, we address *I/O protocol violation bugs* that affect storage device drivers, by proposing an approach (MoIO) for detecting I/O protocol violations at run-time. Failure detection is a fundamental prerequisite for adopting fault-tolerance strategies: if a failing device driver is timely detected, then it can be stopped before executing invalid operations that could corrupt data; moreover, the system administrator can be informed about the anomaly, thus increasing the chances of protecting and recovering data. The detection of protocol violations is challenging, as it requires a *model* of the driver protocol, which should serve as a reference of the correct behavior, for checking whether the actual behavior of the driver deviates from it. However, such a protocol model is difficult to get, since the source code of the device driver may not be available, and the device control logic is often not accurately documented [6].

As discussed in this paper, we address the lack of a reference model by developing a *model inference* technique, which automatically learns a protocol model from failure-free execution traces of the device driver. Then, the model is translated into a lightweight *kernel monitoring module*, which is executed alongside the device driver in order to detect I/O protocol violations. This monitor probes the interactions among the storage device driver, the OS, and the physical devices (e.g., reads/writes to I/O registers), and quickly raises an alarm once an anomalous interaction is detected. We applied the proposed monitoring approach on the SATA/AHCI device driver of the Linux kernel. To evaluate the effectiveness of the approach, we conducted an extensive fault injection campaign, in which the monitor was able to detect storage data corruptions. Moreover, we also conducted long-running tests using a set of several I/O-bound workloads, showing that the I/O monitor is robust against false alarms and has a small performance overhead.

The paper is structured as follows. Section II gives an overview of past studies on fault tolerance for device drivers. Section III discusses in depth the problem of I/O protocol violations, and presents the proposed approach. Sections IV and V discuss the application of the approach on the Linux kernel and experimental results. Section VI closes the paper.

II. Related work

Research on OS reliability has been mostly focused on *tolerating faulty device drivers*. In particular, several *software fault isolation* techniques have been developed to prevent (either in hardware or in software) the propagation of drivers' faults across OS components (e.g., by accessing and corrupting kernel memory), which may cause data loss and the crash of the whole system.

Nooks [9] ensures isolation by confining drivers into a *domain*, using the Memory Management Unit of the CPU. These domains share the same address space of other kernel

components, but different components have different access permissions to pages. In this way, a driver can read all pages, but it is allowed to write only a subset of them. The same authors [10] later proposed a technique to automatically recover from transient drivers' faults, by introducing a *shadow driver* that runs alongside a device driver: when a driver failure is detected, the shadow driver becomes active and replies, on behalf of the faulty driver, to kernel requests, in order to guarantee availability. Subsequent studies further developed these ideas by moving device drivers into user-space processes (thus, running them in unprivileged mode), such as the *microdrivers* approach [11], and the *microkernel* OS architecture (such as Minix3) [12]. Furthermore, the I/O Memory Management Unit has been adopted to isolate kernel data from faults that affect DMA transfers [13], [14].

It is important to note that software fault isolation prevents device drivers from access *data of other OS components*, but it does not prevent device drivers from *corrupting the data managed by them* (e.g., memory and storage data of a storage device driver). Protecting the hardware device from driver's faults is still an open problem. *Guardrail* [15] is one of the few existing solutions, in which a hypervisor protects the hardware device from data races and memory access failures. More specific solutions for storage problems, such as filesystem verifiers (e.g., *fsck*) [16], check the consistency of filesystem *metadata*, but they may not be able to detect silent *data block* corruptions, and can have a significant overhead (and are only run on a periodical basis) [17]. Detecting such corruptions would require end-to-end checks on data block contents', for instance by storing checksums alongside data, and verifying the checksum when retrieving data [18]. Some studies have introduced this kind of checks between the filesystem and the device driver [19], [20]. However, we note that the effectiveness of checksums is limited since *data corruptions are detected only when such data is retrieved for reading*, which may occur only after a long time has elapsed since the corruption. To compensate for this gap, in this paper we investigate an approach for monitoring the individual interactions between the device driver, the hardware device and the OS, and to timely detect incorrect interactions as soon as they take place.

III. The run-time monitoring approach

We propose a run-time monitoring approach for detecting I/O protocol violations of storage device drivers. It monitors how the storage device driver interacts with (i) the physical device, to transmit I/O commands, and (ii) other software components of the OS, to transfer I/O data from and to user applications. The interactions follow a *protocol* that specifies the type and the order of interactions to accomplish I/O operations. Device driver bugs can lead to interactions that violate such a protocol: in the worst case, the device driver can corrupt data on the physical device.

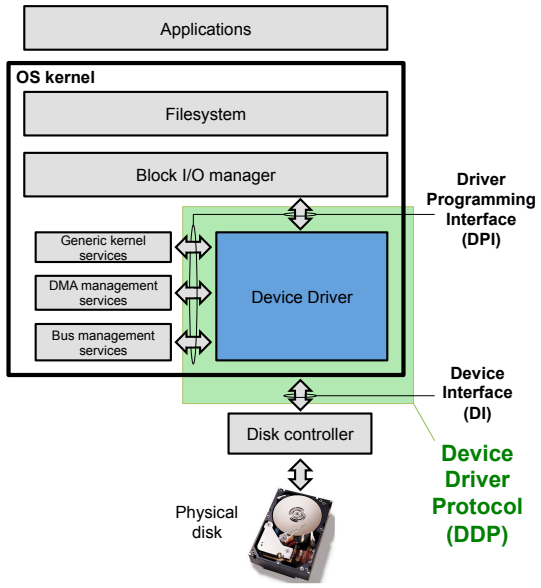


Figure 1. Storage architecture.

A. Background on storage architectures

To understand the nature of device driver interactions, we need to consider the typical architecture of the storage stack (see Figure 1), which includes the following components:

- The **file system**, which provides to the user the abstraction of files and directories, along with their metadata;
- The **block I/O manager**, which provides a block device abstraction to the filesystem. It schedules I/O block read and write requests generated by the filesystem, and manages buffering and caching of disk data on RAM;
- The **generic kernel services**, that is, APIs for dynamic memory allocation, timers, synchronization, and interrupt management, and other services used by developers for supporting I/O transfers in the device driver;
- The **bus management services**, that is, APIs for identifying devices, setting their status and getting information about them (such as the device model, and the address of memory-mapped I/O registers);
- The **DMA management services**, which manage the off-loading of I/O transfers to DMA (Direct Memory Access) controllers (thus relieving the CPU during transfers), and the allocation of DMA memory areas;
- The **device driver**, which provides a software interface for the device to the block I/O manager, and actually controls I/O transfers;
- The **disk controller**, which exposes a set of I/O registers and memory areas to the device driver, and is controlled by the device driver.

Storage I/O starts from *read* and *write* operations made by user applications on the filesystem. These operations flow to the device driver. The device driver then accesses the controller, and uses OS services (e.g., DMA and bus APIs).

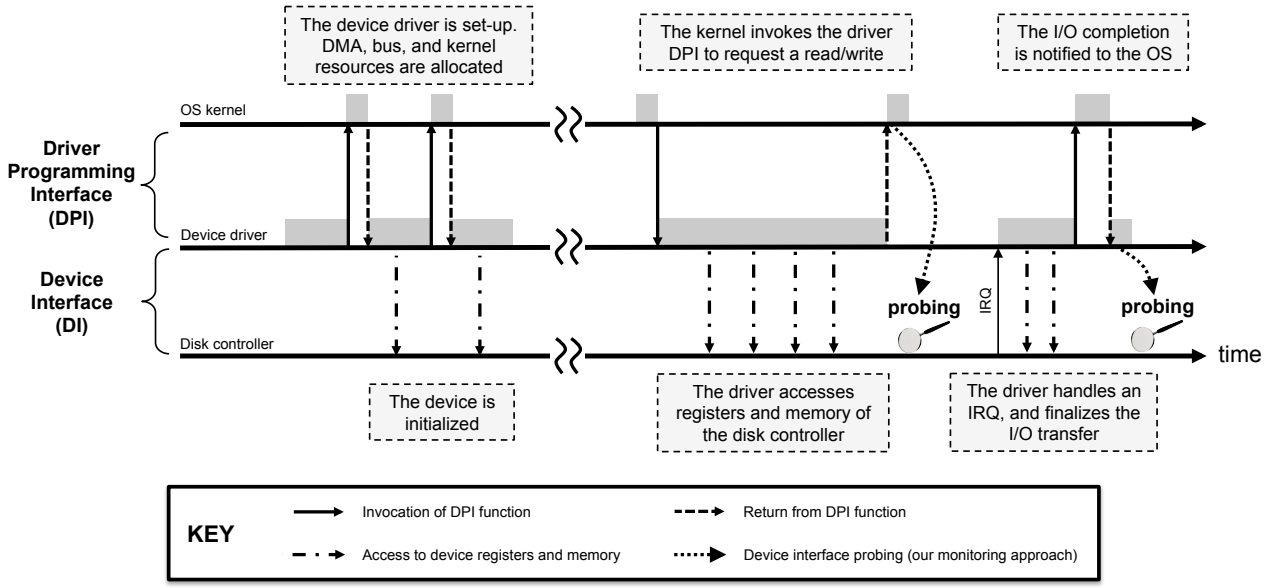


Figure 2. Interactions between the device driver, the disk controller, and the OS kernel.

B. Background on the Device Driver Protocol (DDP)

All the interactions between the device driver and the rest of the storage stack follow the **Device Driver Protocol** (DDP). Overall, I/O operations require a series of interactions between the device driver, the OS and the disk controller. These interactions must follow strict rules for the correct execution of I/O operations. The DDP involves: (i) *software interfaces* of the OS kernel (such as APIs for I/O transfers and for managing kernel resources); and (ii) *hardware interfaces* with the device (such as I/O control and status registers, and interrupts). The software and hardware interfaces are respectively referred to as the **Driver Programming Interface** (DPI) and the **Device Interface** (DI). For instance (Figure 2), when an user application requests an I/O write operation, the OS interacts with the device driver through the DPI, and the device driver reads and writes the DI to actually perform the I/O transfer on the disk.

The DDP can be quite complex, as it involves several layers of the storage stack, both in hardware and in software (Figure 1). On the hardware side, the DDP involves the DI, which is specified by industry standards from organizations such as the ANSI, the ISO, and the IEC. These standards describe several rules for communicating with the device, including the layout of registers and data structures, the format of I/O commands, data, and status information, and the temporal sequence at which this information should be read and written from/to the disk controller. On the software side, the interactions between the device driver and the OS are defined by OS developers in the form of DPI rules. For instance, such interactions can include the sequence of APIs to be called for configuring a DMA buffer, and for initializing a data structure with I/O data or commands.

The goal of our run-time monitoring approach is to detect whether the device driver deviates from the DDP. Detecting such violations requires a reference model of the DDP, which should accurately specify how the device driver is expected to interact with the rest of the storage stack. However, obtaining an accurate model is a challenging problem. On the hardware side, as a matter of fact, industry standards intentionally leave out of scope some aspects of the disk device specifications, which vary across different manufacturers, or even across different products of the same manufacturer (e.g., a specific disk controller can provide registers not required by an industry standard, to enhance performance or to provide more rich functionalities). In many cases, device manufacturers deviate from standard specifications, or may adopt proprietary interfaces not compliant with any standard. On the software side, the DDP is also difficult to model, since the DPI rules between the device driver and the OS are documented in natural language, and in some cases such documentation is lacking, outdated or not consistent with the actual implementation. Even worse, the DPI itself often varies across different OS versions.

C. Overview of the monitoring approach

Given the architecture and the issues discussed above, we design a monitoring approach by taking into account the following requirements:

- **No need for user-provided protocol specifications**: Since the DDP has a cross-layer nature and its specification is not easily available, we do not require the user to provide a specification of the DDP.
- **Applicable on binary-only device drivers**: We do not assume the availability of the source code of the device driver, since the device driver may be only provided in

binary form, for instance by a device manufacturer that wants to protect intellectual property.

- **Low-overhead**: Run-time monitoring should not cause a significant loss of I/O performance, since it would discourage users that do not want to sacrifice performance for improving reliability.

To fulfil these requirements, the MoIO approach adopts a technique to *automatically infer* the DDP model from failure-free execution traces of DDP events. These events are then monitored in production for detecting protocol violations, using the inferred DDP model. Events are collected at driver's interfaces (the DPI and the DI), which can be traced even without access to the source code of the driver. We design the approach to collect only a small amount of information to keep low the performance overhead.

The proposed monitoring approach consists of two phases. The first phase (**model learning and synthesis**) collects execution traces from the OS, and generates a monitoring component (which we call *monitor*) from them. Traces are obtained by inserting low-overhead probes at selected points of the kernel, which intercept and record I/O interactions during the execution of the storage device driver. The traces are turned into a finite state machine (FSM) that summarizes the behavior of the device driver according to the traces. Finally, a monitor is generated from the learned FSM. In the second phase (**run-time monitoring**) the monitor is deployed in production, to detect deviations from the DDP.

The underlying idea of the approach is to automatically generate a behavioral model of the device driver by analyzing its failure-free execution traces. This idea is based on the observation that device driver failures have a *transient* nature, and are triggered by relatively-rare environmental factors that are difficult to reproduce and to debug [21], [10], [12], [22], [23], [24], [25]. Such environmental factors include multi-threading, asynchrony, interrupts, locking protocols, hardware events and virtual memory [21].

For this reason, it is feasible to collect failure-free execution traces before deployment (as failures are unlikely), and use these execution traces to build a "reference model", that will be used at run-time to detect anomalous behaviors (i.e., transient failures) that have not been observed before. The device driver can be executed and traced in a testing environment, by using traditional performance benchmarks or other test workloads. The user can guarantee the absence of errors during the training phase, since he/she has full control on the workload executed and can easily check its (expected) outputs. For example, if the training workload consists in a file copy, the user can check that the copy is equal to the original file. In a similar way, benchmarking tools allow checking the end-to-end correctness of I/O transfers. For example, *IOzone* provides the "+*d*" option for this purpose [26]. Tools for checking the consistency of filesystems and DBs can also be applied.

Technical details about the approach are presented in the following subsections, which discuss the monitoring architecture (III-D) and the model learning algorithm (III-E).

D. Monitoring architecture

The high-level monitoring architecture of MoIO is showed in Figure 3. It includes a *prober* component, which is triggered both at the start and at the end of every I/O operation. When triggered, it records the current contents of the device interface (DI), which include commands and status information for on-going I/O operations. This information from the DI is used to keep track of the "current state" of the DDP. During the *model learning* phase, the *prober* copies the contents of the DI in a trace, to be later processed and synthesized into an executable *monitor* component. During the *run-time monitoring* phase, the *prober* collects again the contents of the DI, and forwards this information to the *monitor* component, which checks whether the current state complies to the DDP model. If this is not the case, the *monitor* raises a warning to notify a DDP violation. For instance, a violation may occur when the device driver does not follow the correct format and sequencing of commands and data according to the DDP (e.g., the device driver writes invalid or out-of-order commands on the DI).

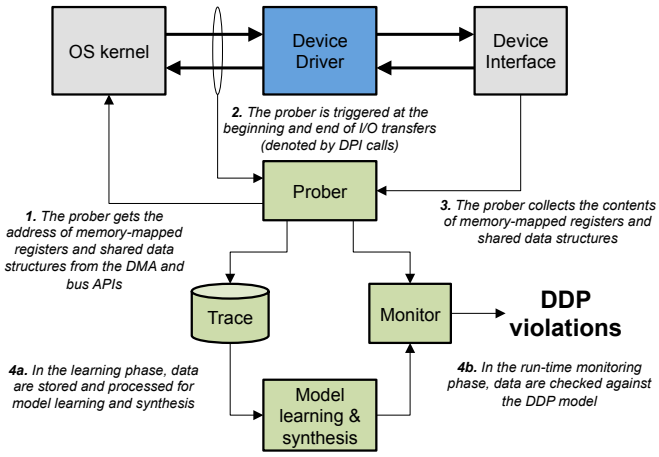


Figure 3. High-level architecture of the MoIO monitoring approach.

As discussed in subsection III-B (see also Figure 2), the device driver interacts with the OS and with the device controller when: (i) the device driver starts an I/O transfer, and (ii) the device driver finalizes an I/O transfer and notifies the OS. On these events (denoted by the "*probing*" marks in Figure 2), the device driver updates the contents of the DI. Therefore, to identify the DDP, our approach collects and analyzes the DI when these events occur. In modern storage controllers (Figure 4), the DI consists of:

- *Memory-mapped registers*, that is, control, data, and status registers of the disk controller that are associated to the physical memory address space. These registers are accessed by the device driver using the same instructions for accessing RAM memory.

- *In-memory data structures* that are shared between the device driver and the disk controller to exchange complex information. For instance, in the case of storage devices, shared data structures are used to provide a list of DMA buffers to the disk controller, or a vector of commands for performing several concurrent I/O transfers. The device driver allocates the data structure in memory (using kernel APIs), and shares it with the disk controller by writing its address on a register.

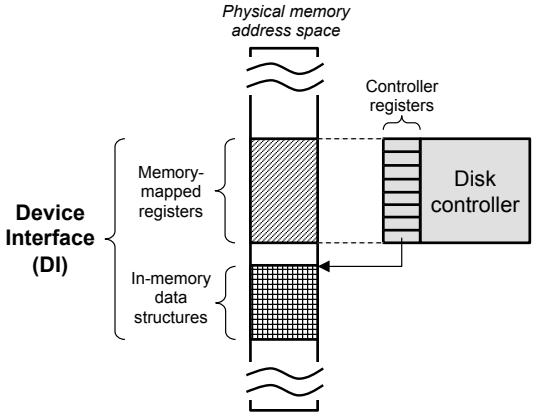


Figure 4. Device interface (DI) inspected by the *prober*.

During the model learning phase (which takes place *before deployment*) we collect, for each I/O transfer, full snapshots of all memory-mapped registers and in-memory data structures, where a trace is a sequence of several such snapshots. During the monitoring phase (*after deployment*, when we are concerned about performance overhead), we only collect a snapshot of few bytes from the DI, which are selected by our model learning algorithm.

The *prober* component is triggered by invocations of the DPI between the OS kernel and the device driver. To trigger the *prober*, our approach leverages probing mechanisms that are available in several commodity OS. These mechanisms allow OS users to dynamically insert *breakpoints* in kernel code, in particular at the invocation and at the return of a kernel function. When the execution of the kernel reaches the breakpoint, it invokes a *handler function* (customizable by OS users) that can collect information about the current execution context: for instance, the handler can inspect the input parameters or the return value of a kernel function invocation. Nowadays, many commodity OSes provide mature dynamic probing mechanisms: the most well-known technologies are *DTrace* for Solaris, Mac OS X and FreeBSD [27], *Kprobes* and *SystemTap* for Linux [28], [29], *Detours* for Microsoft Windows [30], and *VProbes* for VMware ESXi [31]. These probing mechanisms add only a small overhead that is reasonably low for most applications: experiments with heavy I/O workloads showed that the impact on I/O throughput is between 3% and 4.5% in the worst case [31].

In our approach, we use probing mechanisms to intercept the following events (that are emphasized in Figure 2): (i) the beginning of an I/O operation, which occurs when the OS invokes a specific DPI function exported by the device driver; (ii) the completion of an I/O operation, which occurs when the device driver eventually invokes a specific DPI function exported by the OS. Therefore, we insert breakpoints at: (i) the function(s) for starting an I/O transfer (exported by the device driver), and (ii) the function(s) for finalizing an I/O transfer (exported by the OS kernel). The *prober* is triggered right after the execution of the probed DPI functions, when the control flow returns from the invocation. Then, the handler collects the contents of the DI. The *prober* gets the memory addresses of controller registers and data structures by invoking and/or probing the DPI. The address of controller registers is obtained from the bus management APIs, which poll the system bus to query connected devices. The address of in-memory data structures is obtained by probing the invocation of DMA APIs, which are called by the device driver to allocate and share in-memory data structures driver initialization (Figure 2).

E. Model learning and synthesis

The model learning and synthesis phase collects execution traces using the techniques discussed above, infers a model of the DDP from the traces, and generates a monitor to check the driver behavior at run-time. It consists of these steps:

1) *FullTrace* ← *RunSystemWithFullTracing()*
2) *SelectedColumns* ← *FilterColumns(FullTrace)*
3) *FilteredTrace* ← *RunSystemWithPartialTracing(SelectedColumns)*
4) *FSM* ← *ModelLearning(FilteredTrace)*
5) *ModelSynthesis(FSM)*
6) *DDPViolations* ← *RunSystemWithMonitor()*

In the first step of this process, we execute the system under analysis to collect "full" traces, that is, by recording the whole contents of the DI (including all memory-mapped registers and data structures). The raw trace (see the example of Figure 5) contains a sequence of samples, where each sample consists of all the contents of the DI (represented as a vector of bytes) after each invocation of the probed DPI. Moreover, each sample includes the DPI function that triggered the prober (e.g., the hypothetical *issue* and *complete* DPI functions in Figure 5).

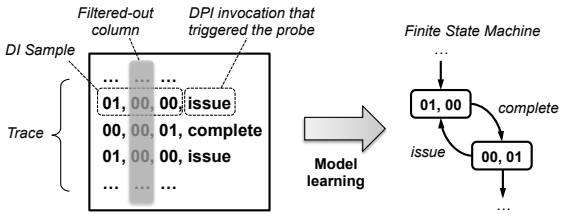


Figure 5. Simplified example of trace.

The initial tracing needs to sample the whole contents of the DI since, before the model learning process, we still do not know the DDP and which parts of the DI are relevant for monitoring purposes. By analyzing this initial trace, our approach learns about which types of information are included in the DI (e.g., command codes, bitmasks, addresses, etc.) and selects a subset of DI's contents, which will be the focus of the subsequent steps.

In fact, we note that not every information from the DI is useful: for instance, we are not interested at analyzing registers that are never modified during I/O transfers (e.g., registers only written during the device initialization). Moreover, we want to avoid noisy and highly-variable information that have little influence on the DDP, such as memory addresses (which are non-deterministically selected by the dynamic memory allocator). Including noisy information would inflate the learned model (thus potentially increasing the performance overhead), and would make it prone to false alarms when there are small variations in I/O transfer patterns. Instead, the most valuable information for the DDP model is the sequence of commands and status information that is exchanged between the device driver and the disk controller. Focusing on this information makes the model compact and robust against small variations in I/O patterns (e.g., the I/O command codes are still the same regardless of allocated memory addresses).

The *FilterColumns* procedure inspects raw traces to locate useful information in the DI. Each sample in the trace is split into a *sequence of bytes*. The set of bytes at the same position of each sample (e.g., the third byte of each sample) is referred to as *column* of the trace. The output of this procedure is the list of columns of the trace with relevant information. For instance, in Figure 5, *FilterColumns* removes the second column, and retains the other two for further analysis. *FilterColumns* identifies:

Constants: Columns are first analyzed independently from each other. For each column, we analyze the number of different values that are assumed by the bytes in the column, and *remove columns that are constant* across the whole trace.

Large fields: The remaining columns are further analyzed, to coalesce adjacent columns that are part of the same information (e.g., a group of several bytes that form a command code). Two or more columns are *coalesced into a new multi-byte column* when: (i) they are located at adjacent positions (e.g., two subsequent bytes from an in-memory data structure), and (ii) there is at least one sample in the trace where all the bytes in the group vary from the homologous bytes in the precedent sample. For instance, if two adjacent bytes both change their values between two consecutive samples, then their columns are grouped.

Bitmasks: Columns that represent bitmasks (e.g., bits with the state of a set of I/O ports) are denoted by variations that involve only involve one bit at a time. We identify them by: (i) comparing, for each byte in each sample, the number of bits that have varied compared to the homologous byte in the precedent sample; (ii) identifying columns where, in most of the samples (e.g., 90% or more), the variations only involve one bit. The margin of tolerance accounts for sporadic cases when several bits vary almost at the same time (e.g., the completion of concurrent I/O transfers). These columns are rewritten by replacing bitmask values with *the number of bits that change between consecutive samples*, which is the most useful information from the point of view of the DDP protocol, and allows to simplify the DDP model.

Addresses: To deal with columns that are prone to variations across different executions (e.g., columns with memory addresses), we *remove the columns that exhibit a high number of different values* (i.e., the cardinality of the column). This strategy is based on the observation that command and status codes (and similar information) are unlikely to exhibit very large cardinality. Since such fields can assume few tens of different values at most, it is safe to discard columns that assume more than one hundred values. Another strategy is to remove columns that only contain memory addresses: these columns can be identified by also tracing memory areas allocated to the driver (by probing memory allocation APIs) and comparing their addresses with the values in the columns.

Once relevant columns have been selected, the system under analysis is again executed (*RunSystemWithPartialTracing(...)*), but this time only "partial" traces are collected, i.e., we collect only the bytes from the DPI that are selected by the previous procedure (*SelectedColumns*). We re-execute the system since partial tracing has much lower overhead than full tracing, and thus is less likely to perturb the execution of the system (the "probe effect" [32]). While full tracing is useful to identify data types in the DI (commands, bitmasks, addresses, ...), partial tracing is more suitable to collect accurate traces of the dynamic behavior of the device driver (e.g., by preserving the timing of events and the sequencing of commands). Columns in the partial trace are coalesced and converted to bitmask counts.

From the filtered trace, we generate an FSM (*ModelLearning*). Each distinct vector of bytes in the trace is turned into a state of the FSM, and the DPI function that triggered the probe is turned into a transition between the current state and the previous one. In the example of partial trace in Figure 5, two states are generated from the two distinct vectors $(01, 00)$ and $(00, 01)$, with two transitions *issue* and *complete* that connect them. Finally, the FSM is synthesized into a *monitor* component. The *monitor* collects selected bytes from the DI, in the same way of partial tracing. In addition, the *monitor keeps track of the state of the device driver on the FSM, and detects a DDP violation when the current state is not compliant to the FSM*.

To tune the duration of tracing runs, the user can perform

several learning trials with increasing duration. At each trial, the resulting monitor can be tested by running again the training workload, and checking the rate of false alarms. When the training workload duration is long enough to provide sufficient information on the device driver's behavior, the rate of alarms abruptly drops to zero.

IV. A CASE STUDY ON THE LINUX KERNEL

In the following of this study, we apply the proposed monitoring approach on the SATA/AHCI disk device driver of the Linux kernel. Both Linux and SATA/AHCI are mature, advanced, and popular technologies. Needless to say, Linux is today a predominant OS for business-critical servers, and has evolved to support a wide range of storage devices, protocols and filesystems. Moreover, SATA disks are today the most popular and reliable disks for desktop and nearline storage [4], and AHCI is a de-facto standard for managing SATA disks. It should be noted that SATA/AHCI is a very complex case study, which involves all the subtleties that are typical of real-world device drivers. Suffice to say that, due to the complexity of storage devices, almost all other studies on device drivers' reliability did not consider storage, but focus on simpler devices (such as soundcards and ethernet cards). We here provide a brief introduction to SATA and AHCI, and technical details about its practical application.

A. Overview of SATA and AHCI

Serial ATA (SATA) is a standard for the communication with mass storage devices (e.g., magnetic tapes, hard disks, optical disks) that leverages on high-speed serial transmission [33]. The SATA standard introduced several enhancements for storage performance and error detection. The key component is the *Frame Information Structure* (FIS), a frame that encapsulates information (commands and controls) exchanged between the controller and device; in turn, that frame is encapsulated within another frame, which bears flow control and error-detection codes.

The *Advanced Host Control Interface* (AHCI) is a hardware/software interface by Intel [34] to provide easier and more flexible access to SATA devices. The AHCI standard defines all the memory-mapped registers and in-memory data structures (see sec. III-D) that allow communication between the OS kernel and the storage device. Furthermore, AHCI supports advanced features of SATA disks, such as *Native Command Queuing* (NCQ) and *hotplugging*.

Figure 6 gives a simplified view (both hardware and software) of an SATA/AHCI storage stack architecture. In AHCI, the disk controller is named *Host Bus Adapter* (HBA), and can handle up to 32 ports. A port is a physical part of the controller where SATA devices are attached, which can be individually controlled by the device driver. Moreover, each port can handle up to 32 commands issued by the driver, which are written on the *command list*, an in-memory data structure. The device driver issues a command by writing a *command FIS* (including the command op-code, the logical disk block addresses to be read or written, etc.); then, the HBA fetches the FIS, and starts a data transfer; on command completion, the HBA raises an interrupt and writes on another in-memory area, the *Received FIS Structure*, to notify the device driver about the status of the transfer.

An AHCI HBA is a *Peripheral Component Interconnect* (PCI) controller, which exposes a set of memory-mapped *PCI Configuration Space* registers for the initialization and configuration of AHCI devices. Moreover, the AHCI HBA exposes a set of *per-port* registers (i.e., registers replicated for each port). Figure 6 shows some of the most important registers, which include:

- *Generic Host Control* (GHC): describes the capabilities and controls the behavior of the HBA (e.g., which ports the HBA exposes, and which capabilities the HBA supports, such as NCQ);
- *Port Command List Base* (CLB): points to the command list of a specific port;
- *Port FIS Base Address* (FB): points to an in-memory area with the received FIS of a port;
- *Port Interrupt Status* (IS): the interrupt status of a specific port;
- *Port Serial ATA Active* (SACT): a bitmask with the status of each NCQ command entry (e.g., bit at position '3' is set if command entry '3' has been issued).

B. Applying the monitoring approach to Linux SATA/AHCI

To support SATA/AHCI disks, Linux uses a device driver splitted in two kernel modules: *ahci* and *libahci*. When loaded, the *ahci* module requests and maps all memory regions for the HBA, registers an interrupt handler, and resets the HBA. The *libahci* module implements low-level routines to communicate with the HBA. These modules interact with the *libATA* [35] library, which provides a kernel interface for all ATA and SATA device drivers.

To apply the proposed monitoring approach, we have to identify the DPI functions for starting and completing I/O transfers. For the SATA/AHCI driver, these functions are the *ahci_qc_issue* function, which issues a command to the HBA by writing a *command FIS* and registers of the HBA, and the *ata_scsi_qc_complete*, which is a callback provided by libATA, and which is invoked during interrupt handling.

When these two probe points are triggered during execution, we collect a snapshot of memory-mapped registers and in-memory data structures that belong to the device driver and to the HBA. We obtain the address of memory-mapped HBA registers using the PCI kernel APIs (e.g., *pcim_iomap_table*), and the address of in-memory, DMA-mapped areas (e.g., *command list* and the *received FIS*) by probing the DMA APIs when *ahci* is loaded (e.g., *dma_alloc_coherent*). We use SystemTap [29] in order to probe the *issue* and *complete* functions, and to inspect HBA registers and in-memory data structures.

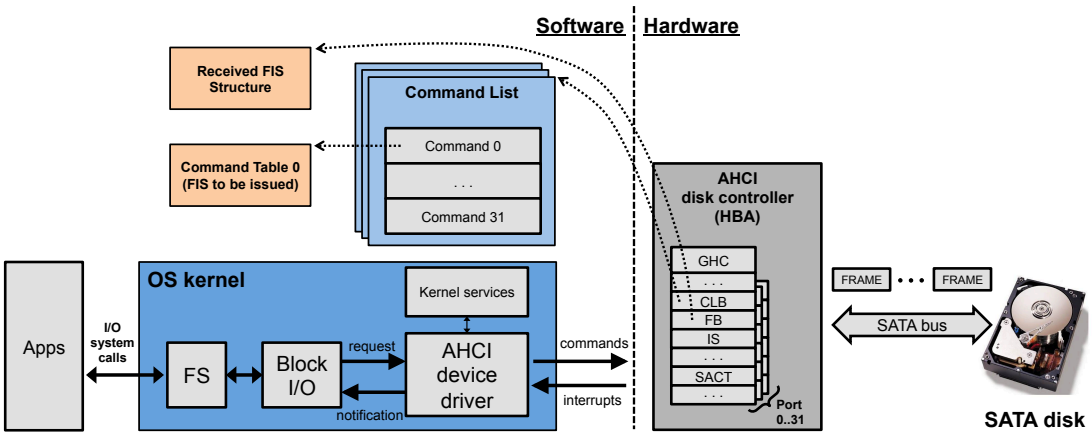


Figure 6. SATA/AHCI architecture.

V. Experimental evaluation

We evaluate our monitoring approach in the context of the SATA/AHCI device driver of the Linux kernel (presented in sec. IV). We consider three aspects for the evaluation: (i) the ability to detect protocol violations, (ii) the amount of false positives, and (iii) the I/O performance overhead. These three aspects are evaluated in two sets of experiments: (i) fault injection experiments, in which we inject faults in the device driver to force failures (including protocol violations) and analyze the *coverage* of the monitor at detecting them; (ii) fault-free experiments, in which we run the monitor under heavy workloads, and analyze the rate of *false positives* (i.e., alarms generated when there are no faults), and the *performance overhead* (i.e., comparing the I/O throughput with and without the monitor).

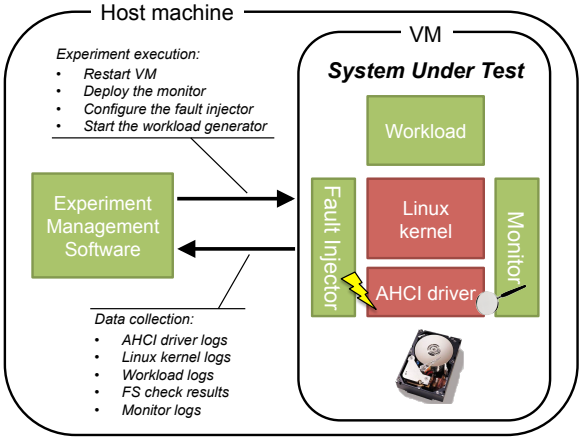


Figure 7. Fault injection setup.

A. Fault injection experiments

We performed fault injection experiments on a Linux system (which we refer to as the *System Under Test* (SUT)) that is based on the Fedora distribution version 21, and on the Linux kernel version 3.19. Experiments are orchestrated by an *Experiment Management Software* (EMS), which configures the SUT, including the workload and the faultload of the experiment, and collects data for later analysis.

Figure 7 shows the experimental setup and the workflow. The SUT is executed in a *virtual machine*, while the EMS is executed in the physical machine that hosts the virtual machine. This separation is more and more adopted in fault injection experiments in OS [36], [37], [38], and is necessary in order to protect the Experiment Management Software from the faults injected into the SUT: when using a virtual machine, the injected faults remain isolated within the SUT, thus allowing the EMS to correctly save the data from the current experiment, and to start the subsequent one. We use VMware Workstation 11 to run the SUT within a virtual machine, which fully emulates a SATA AHCI disk controller (i.e., the disk controller of the virtual machine is managed and accessed by the Linux kernel in the same way of a physical disk controller, using the *ahci* device driver) [39].

Commands for executing experiments (e.g., to start the workload, and to configure the fault injector) are sent through an SSH connection, and experimental data (e.g., logs from the OS, from the monitor, and from user-space processes) are collected from both a virtual serial connection and an SSH connection. Experimental data are analyzed to identify whether the injected fault caused a failure of the SUT. Possible outcomes of the experiments are:

- The *crash* or *stall* of either the OS or of the workload. They are identified by crash messages, and by checking the responsiveness of the SUT at the end of the experiment (e.g., the SUT completes all experimental steps and responds to commands from the EMS);
- *Data corruptions*, that is, executions that neither crash nor stall, but where the data are corrupted at the end of the experiment. They are identified by (i) checking the contents of the output files produced by the workload (by comparing them to fault-free executions), and (ii) checking the consistency of filesystem data structures

(such as damaged directory trees and "orphaned" files) using the *fsck* filesystem verification tool;
- *No failure* of the SUT. This outcome happens when the injected fault does not propagate within the SUT (e.g., corrupted data is never accessed, or is overwritten by correct data), and is identified by the lack of failures.

Our analysis is focused on I/O protocol violations that cause *data corruption* failures. These failures are subtle and not detectable by simply checking for crashes or stalls; our monitor fills this gap by detecting I/O protocol violations. We evaluate the *coverage* of failure detection, which is *the percentage of experiments where data corruption is detected, among all experiments with data corruption failures*.

We perform three fault injection campaigns by using the following well-known I/O workloads [40], [41]:

- **IOzone**: A filesystem benchmarking tool. It generates a variety of file operations, using several read and write patterns (e.g., sequential, random, repeated, strided), sizes (e.g., both small and large record and file sizes), and APIs (e.g., memory-mapped, asynchronous I/O);
- **Postmark**: An I/O benchmark that emulates a large email server, by performing a mix of data- and metadata-intensive operations on a pool of random text files. It first creates the pool of files with uniformly distributed sizes, and then it performs a sequence (namely *transaction*) of randomly selected I/O operations (e.g., file creation, deletion, read, and append);
- **SQLite**: a software library that implements a serverless transactional SQL database engine. It is exercised by performing several *insert* and *select* queries.

We generate the *monitor* component by running the SUT using the IOzone workload. Then, we use the monitor trained with IOzone for failure detection in all the three fault injection campaigns. In this way, we are considering both (i) the case when the workload for training the monitor is representative of the workload in production (fault injection experiments under IOzone), and (ii) the case when the workload in production differs from training (experiments under Postmark and SQLite). Executing the IOzone workload for five minutes was sufficient to train a robust monitor.

We inject faults in the SATA/AHCI device driver during the execution of the SUT. We adopt the fault injector developed by Ng and Chen for testing the Rio File Cache in FreeBSD [42], and later ported to Linux and Minix in subsequent studies on OS fault tolerance [21], [12]. This fault injector emulates software bugs that are common in OS code, according to field failure data [43]. The injector can emulate *assignment* faults (i.e., incorrect source or destination in assignment instruction), *control* faults (i.e., incorrect logical condition in loop or branch), *parameter* faults (i.e., incorrect parameter in a function call), *omission* faults (i.e., a missing instruction, by removing it), and *pointer* faults (i.e., an incorrect memory pointer computation). Faults are injected at run-time by replacing the original instructions with corrupted ones in the binary code of the driver [44]. We perform 210 fault injection experiments, by randomly selecting the fault type and location at each experiment.

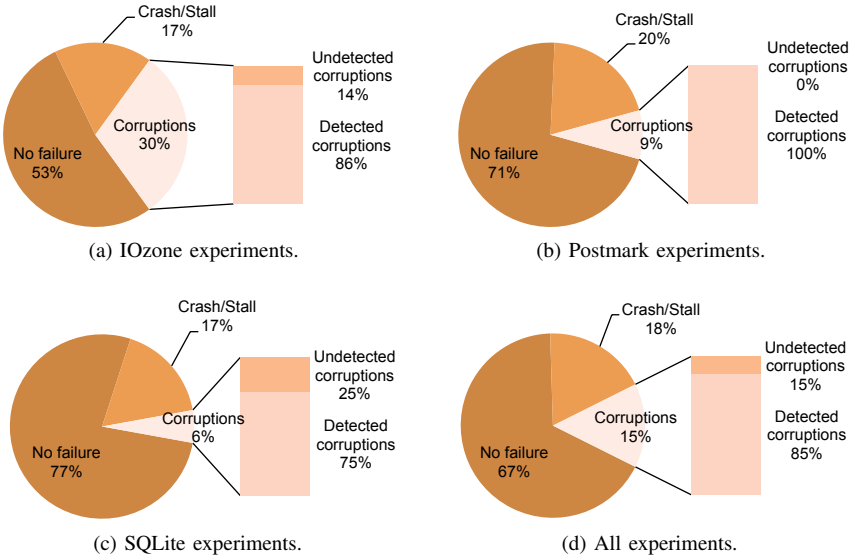
From the experimental data, we identified the outcome of the injected fault (crash/stall, data corruption, no failure) and, in the case of data corruption failures, the outcome of failure detection. Figure 8 summarizes both failure modes and failure detection. Most of the cases resulted in no failures, since the injected fault did not impact on the device driver (e.g., the fault results in corrupted memory that is overwritten or ignored by the device driver)—this is a known phenomenon that is often observed in fault injection experiments [45]. Experiments also include data corruption failures, that affect both application data (causing incorrect outputs) and the filesystem (such as, invalid partition information; orphaned inodes; corrupted superblocks).

The monitor detected data corruptions in 85% of cases (Figure 8d). In these experiments, faults caused the device driver to omit a write on the DI or lead to incorrect writes; they then resulted in I/O transfers that were not completed, and to an incorrect sequencing of events that was detected by the DDP model. Undetected corruptions can be attributed to the relative simplicity of the monitor, which trades-off less important information about the DDP (since some fields of the DI are filtered-out and excluded from monitoring, as discussed in sec. III-E) in order to be lightweight and robust against noise (as discussed in the next subsection).

It must also be noted that the failure detection coverage of the monitor is still high even when the workload for training the monitor is different than the workload of the experiments (i.e., the coverage in Figures 8b and 8c is comparable to the coverage in Figure 8a). This can be explained by the layered architecture of the storage stack (Figure 1) between user applications and the device driver: the variations across workloads (e.g., in terms of number and size of files and of I/O system calls) are not perceived by the device driver, since the filesystem and the block I/O manager (which performs I/O buffering, caching and scheduling) turns them into a regular sequence of I/O interactions. This behavior allows to use the monitoring approach even if there is a lack of a representative workload of the production environment, since using a generic (but enough comprehensive) benchmarking tool, such as IOzone, suffices to train an effective monitor.

B. Performance and long-running experiments

We evaluate the robustness of the monitoring approach to false positives, by executing the SUT for a long time in the absence of faults. We perform three long-running experiments, by repeatedly executing the IOzone, Postmark, and SQLite workloads for one day. Again, we train the monitor using IOzone, and use this monitor in all three experiments. To assure a fault-free execution, we check the correctness of outputs of the workload at each iteration



(a) IOzone experiments.

(b) Postmark experiments.

(c) SQLite experiments.

(d) All experiments.

Figure 8. Distributions of fault injection outcomes, and failure detection coverage.

and do not inject any fault during the experiment. In these conditions, the monitor *should not* detect any failure, thus any DDP violation raised by the monitor is a false positive.

The long-running experiments gave a promising result: no false alarms were raised. This result must be interpreted with caution, since our monitor cannot assure freedom from false positives. In general, any method based on dynamic program analysis can be affected by false positives, since execution traces of large software are not a comprehensive representation of all possible device driver executions. However, the model learning technique has been able to extract only minimal and stable features of the device driver protocol, that are very likely to hold under fault-free executions. We avoid false positives by filtering the parts of the device interface that are prone to very high variability and that would add little to the failure detection power. This allows to achieve a reasonable trade-off between false positives, false negatives, and overhead for practical applications.

Finally, we evaluated the impact of run-time monitoring on performance. We measured the execution time of the IOzone, Postmark, and SQLite workloads, respectively *without* and *with* monitoring. Figure 9 shows the average and the standard deviation of the execution time, that have been evaluated from 20 repeated executions of each workload. Monitoring introduces a very small overhead (mostly due to the overhead of dynamic probing mechanisms), since the average execution time increases only by 2.97% in the worst case (Postmark), and the differences of average execution time are not statistically significant according to a t-test.

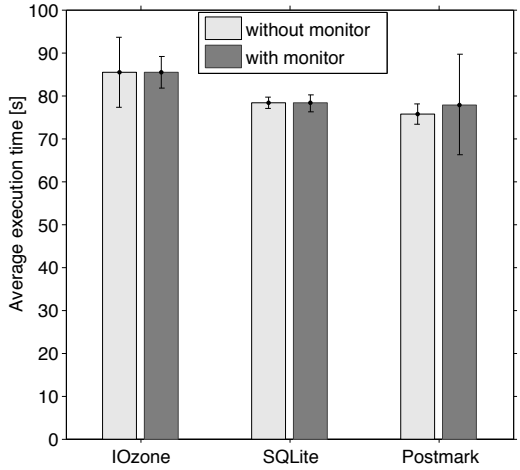


Figure 9. Performance overhead of the monitoring approach.

VI. Conclusion

We addressed the problem of detecting I/O protocol violations of storage device drivers at run-time. We proposed a monitoring approach based on a model inference technique, which automatically learns the device driver protocol without relying on documentation or source code. This approach has been applied on a complex, real-word case study (the SATA/AHCI driver of the Linux kernel), showing that it is a promising solution with a good failure detection coverage, a low rate of false positives, and small overhead.

Acknowledgment

This work was supported by the COSMIC public-private laboratory, projects SVEVIA (PON02_00485_3487758), MINIMINDS (PON02_00485_3164061) and DISPLAY (PON02_00485_3487784), and the TENACE PRIN project (n. 20103P34XC), funded by the Italian Ministry of Education, University and Research.

REFERENCES

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *SOSP '01*.

[2] A. Ganapathi, V. Ganapathi, and D. A. Patterson, "Windows XP Kernel Crash Analysis," in *LISA'06*.

[3] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: Ten years later," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, 2011.

[4] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, "Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics," *ACM Transactions on Storage*, vol. 4, no. 3, 2008.

[5] W. Jiang, C. Hu, A. Kanevsky, and Y. Zhou, "Don't blame disks for every storage subsystem failure," *;login:*, vol. 33, no. 3, 2008.

[6] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser, "Dingo: Taming device drivers," in *EuroSys'09*.

[7] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with ddt," in *USENIXATC'10*.

[8] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: Testing drivers without devices," in *OSDI'12*.

[9] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *SOSP '03*.

[10] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Transactions on Computer Systems*, vol. 24, no. 4, 2006.

[11] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, "The design and implementation of micro-drivers," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, 2008.

[12] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Fault isolation for device drivers," in *DSN '09*.

[13] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider, "Device driver safety through a reference validation mechanism," in *OSDI'08*.

[14] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in linux," in *USENIXATC'10*.

[15] O. Ruwase, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Guardrail: a high fidelity approach to protecting hardware devices from buggy drivers," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, 2014.

[16] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*. Wiley, 2013.

[17] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. K. Mckusick, "Ffsck: The Fast File-System Checker," *ACM Transactions on Storage*, vol. 10, no. 1, 2014.

[18] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, 1984.

[19] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Improving file system reliability with I/O shepherding," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, 2007.

[20] J. N. Herder, D. C. Van Moolenbroek, R. Appuswamy, B. Wu, B. Gras, and A. S. Tanenbaum, "Dealing with driver failures in the storage stack," in *LADC'09*.

[21] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, 2003.

[22] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum, "Failure resilience for device drivers," in *DSN '07*.

[23] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot; a technique for cheap recovery," in *OSDI'04*.

[24] A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano, "Effective fault treatment for improving the dependability of COTS and legacy-based applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 4, 2004.

[25] R. Natella and D. Cotroneo, "Emulation of transient software faults for dependability assessment: A case study," in *EDCC'10*.

[26] "IOzone Filesystem Benchmark," http://www.iozone.org/, accessed: 2015-08-01.

[27] B. Gregg and J. Mauro, *DTrace: Dynamic Tracing in the Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[28] W. Cohen, "Gaining insight into the Linux kernel with Kprobes," 2005, http://www.redhat.com/magazine/005mar05/features/kprobes/.

[29] ——, "Instrumenting the Linux Kernel with SystemTap," 2005, http://www.redhat.com/magazine/011sep05/features/systemtap/.

[30] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," in *Usenix Windows NT Symposium*, 1999.

[31] M. Carbone, A. Kataria, R. Rugina, and V. Thampi, "VProbes: Deep Observability into the ESXi Hypervisor," *VMware Technical Journal, Summer*, 2014.

[32] J. Gait, "A probe effect in concurrent programs," *Software: Practice and Experience*, vol. 16, no. 3, 1986.

[33] Serial ATA International Organization, *Serial ATA Revision 3.0*, www.sata-io.org.

[34] Intel Corporation, *Advanced Host Controller Interface for Serial ATA*, http://www.intel.com/content/www/us/en/io/serial-ata/ahci.html.

[35] LibATA. Linux ATA wiki - Main Page. https://ata.wiki.kernel.org/index.php/Main_Page.

[36] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato, "D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology," in *CCGRID'10*.

[37] I. Irrera, J. Durães, H. Madeira, and M. Vieira, "Assessing the Impact of Virtualization on the Generation of Failure Prediction Data," in *LADC'13*.

[38] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "No PAIN, No Gain? The Utility of PArallel Fault INjections," in *ICSE'15*.

[39] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing Virtualization to the x86 Architecture with the Original VMware Workstation," *ACM Transactions on Computer Systems*, vol. 30, no. 4, 2012.

[40] A. Traeger, E. Zadok, N. Joukov, and C. Wright, "A Nine Year Study of File System and Storage Benchmarking," *ACM Transactions on Storage*, vol. 4, no. 2, 2008.

[41] K. Kanoun, Y. Crouzet, A. Kalakech, A. Rugina, and P. Rumeau, "Benchmarking the Dependability of Windows and Linux using PostMark™ Workloads," in *ISSRE'05*.

[42] W. T. Ng and P. M. Chen, "The design and verification of the rio file cache," *IEEE Trans. Comput.*, vol. 50, no. 4, 2001.

[43] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems," in *FTCS'91*.

[44] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *EDCC'12*.

[45] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An Empirical Study of Injected versus Actual Interface Errors," in *ISSTA'14*.