Run-Time Detection of Protocol Bugs in Storage I/O Device Drivers

Domenico Cotroneo, Luigi De Simone, Roberto Natella

***Abstract*—Protocol violation bugs in storage device drivers are a critical threat for data integrity, since these bugs can silently corrupt the commands and data flowing between the OS and storage devices. Due to their nature, these bugs are notoriously difficult to find by traditional testing. In this paper, we propose a run-time monitoring approach for storage device drivers, in order to detect I/O protocol violations that would otherwise silently escalate in corruptions of users' data. The monitoring approach detects violations of I/O protocols by automatically learning a reference model from failure-free execution traces. The approach focuses on selected portions of the storage controller interface, in order to achieve a good trade-off in terms of low performance overhead and high coverage and accuracy of failure detection. We assess these properties on three real-world storage device drivers from the Linux kernel, through fault injection and stress tests. Moreover, we show that the monitoring approach only requires few minutes of training workload, and that it is robust to differences between the operational and the training workloads.**

***Index Terms*—Device drivers; Storage failures; Run-time monitoring; Linux kernel; Fault injection.**

I. Introduction

Data are a valuable asset, and assuring their reliability is an important requirement for IT systems. Significant investments are made into technologies and practices for preserving data, including RAID storage, redundant database servers and data paths, sophisticated filesystems, I/O virtualization, regular backups and disaster recovery plans [1], [2]. Nevertheless, faulty device drivers, which are among the most bug-prone OS components [3]–[5] and a frequent source of storage failures [6], [7], still represent a significant threat for the storage stack due to the shortcomings of failure detection mechanisms against software bugs.

In particular, *protocol violation bugs* are the ones that violate the protocol between the software and hardware parts of the storage stack, by misinterpreting the device state or sending incorrect commands to the device. These bugs are quite problematic since they are able to elude traditional testing due to their transient behavior (e.g., they are triggered under very specific states of the hardware and of the OS), which makes it difficult to achieve good test coverage [8], [9]. Moreover, protocol bugs represent a large part of device bugs, as showed by Ryzhyk et al. [10] in the context of the Linux kernel, where they represented 38% of the total. More important, protocol violations have the potential to cause undetected effects. The OS and the hardware focus on detecting generic failure symptoms, such as an access to an invalid memory address, but protocol violations do not necessarily lead to such symptoms. The lack of detection can lead to silent corruptions of users' data, thus exacerbating the cost of software failures.

In this paper, we propose a novel approach for detecting I/O protocol violations in storage device drivers, by monitoring at run-time the interactions between the driver and the hardware device controller. The purpose of the run-time monitor is to detect device driver failures in a timely manner. This solution is meant both to users and engineers of high-availability storage systems, including: administrators and end-users of the system, which need to get alarms about the onset of data corruptions in order to trigger recovery strategies (such as, to stop the faulty machine, and to check for data integrity and recover a recent backup if necessary); kernel developers, which may want to offer a ready-to-use monitoring module as companion of their device driver, in order to ease system administrators and end-users at detecting failures and at providing information for debugging; and vendors of full-stack storage solutions, which may want to integrate additional high-reliability features in their offerings.

The approach addresses the challenging problem of defining a *reference model* of the (correct) driver protocol, in order to point out deviations of the actual driver's behavior from it. Indeed, defining such model is a cumbersome and difficult task, since I/O protocols are complex (e.g., due to concurrency and to their mixed hardware/software nature), and since the device controller may differ from public protocol specifications (e.g., I/O device standards) in subtle and undocumented ways [10]. Therefore, our approach consists in automatically learning a reference model from traces of execution of the device driver under failure-free conditions, by probing how the device driver interacts with the storage controller (e.g., on memory-mapped registers) and with the OS kernel (e.g., invocations of I/O APIs). Then, it generates a kernel module that acts as a *run-time monitor* to check for violations from the learned protocol. The approach focuses on the interfaces of the device drivers, and it only requires limited knowledge of the device driver API and key data structures.

We evaluate this approach on three real storage device drivers of the Linux OS. We analyze the ability of this OS to detect storage data corruptions through an extensive fault injection campaign. More in particular, during the experimental campaign, we addressed the following questions:

▹ ***Monitoring coverage***. Is the learned monitor actually able to detect data corruptions? Does the approach improve detection with respect to existing error detection checks in the I/O stack?

Increasing the detection coverage is a key goal for systems with high reliability requirements, in which even a single case of data loss due to silent corruption has a significant cost.

▹ ***Monitoring accuracy***. The run-time monitoring approach is based on dynamic analysis, in which the monitor is trained only with a finite subset of the many possible execution traces of the driver. Moreover, there will be inevitable differences between the training workload and the actual workload in production environments. Can the false positive rate be kept low enough for practical purposes, despite such limited training traces? How much training is necessary to get an accurate-enough monitor? Having a low volume of false positives, and a quick and robust training are important properties of the approach to be usable.

▹ ***Side effects***. The run-time monitor must instrument the device driver in order to collect state information, but the instrumentation may slow down I/O operations (i.e., performance overheads) and even cause a *probe effect* (e.g., alterations of the driver's behavior due to different timings of events). How to keep the performance overhead low, even under high-volume I/O workloads? How to prevent the *probe effect* from causing alterations that are mistaken by the monitor as protocol violations (i.e., false positives)?

The experimental results pointed out that the monitoring approach can achieve a high coverage (by detecting 86% of data corruption failures), and a good complementarity with other error checks in the kernel (as 26% of the data corruption failures were only detected by the proposed approach). Moreover, the monitoring approach exhibited no false positives during week-long stress tests. Interestingly, both the high coverage and high accuracy of the approach are preserved even when the operational workload differs from the training workload: a closer analysis of the driver API revealed that I/O access patterns at that level are insensitive to workload variations at the user-space level, and that these patterns can be learned by our approach within few minutes of training workload. Finally, the performance overhead was negligible in most of the experimental scenarios.

The paper is structured as follows. Section II provides background on the storage I/O stack and on device driver protocols. Section III presents the proposed monitoring approach. Sections IV and V present the case studies and the experimental results. Section VI discusses the evaluation methodology and the practical implications of the proposed approach. Section VII reviews the related work. Section VIII closes the paper. The appendix provides more detailed technical information on the case studies.

II. Technical background

The I/O protocol defines the types and the order of interactions between the device driver and the other elements of the storage I/O stack. In this work, the focus is on the detection of device driver bugs that violate the I/O protocol.

A typical I/O stack is depicted in Fig. 1. On the one hand, the device driver interacts with the *OS kernel*. The OS kernel provides *system calls* to user applications to read and write files on the storage. The system calls are served by the filesystem, which uses the block I/O subsystem for transferring individual file blocks (e.g., this subsystem schedules I/O requests and caches recent I/O blocks). In turn, the block I/O subsystem invokes the device driver to transfer the blocks. Moreover, the device driver also interacts with the kernel through generic kernel APIs for managing the I/O bus (e.g., identifying devices and getting information such as the device model and state) and for performing DMA (Direct Memory Access), i.e., for off-loading to a dedicated chip the task of copying I/O data between the controller and memory, and for allocating memory for such transfers.

On the other hand, the device driver interacts with the *storage controller*, i.e., the electronic hardware interface of the device. The controller is connected to the system bus, and it can be accessed by the CPU and the device driver by means of *memory-mapped I/O registers*, i.e., registers of the controller that are associated to the memory address space, and that are accessed in the same way as physical memory locations using the same CPU instructions (e.g., loads and stores). The addresses of memory-mapped I/O registers are obtained by the driver through the bus I/O API. These registers expose the state of the device, and can be used by the device driver to write commands and to read/write I/O data. Moreover, memory-mapped registers can point to physical memory areas for large data transfers, such as DMA memory areas.

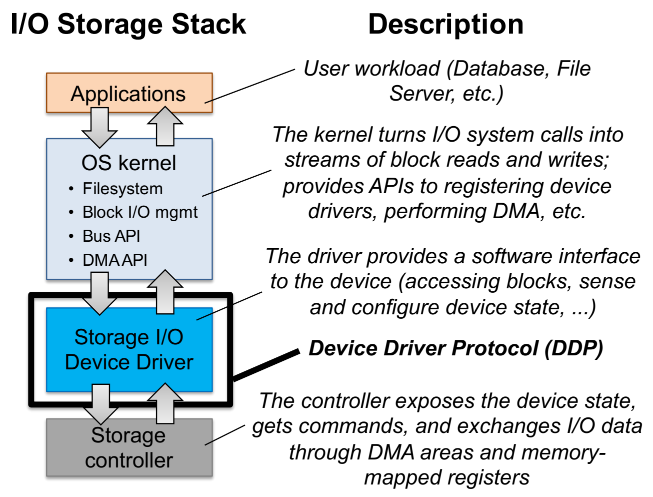

I/O Storage Stack
Description
Applications
User workload (Database, File Server, etc.)
OS kernel
• Filesystem
• Block I/O mgmt
• Bus API
• DMA API
The kernel turns I/O system calls into streams of block reads and writes; provides APIs to registering device drivers, performing DMA, etc.
Storage I/O Device Driver
The driver provides a software interface to the device (accessing blocks, sense and configure device state, ...)
Device Driver Protocol (DDP)
Storage controller
The controller exposes the device state, gets commands, and exchanges I/O data through DMA areas and memory-mapped registers

Fig. 1. Overview of the I/O storage stack.

The protocol that defines the interactions between the device driver and the rest of the storage stack is referred to as the **Device Driver Protocol** (DDP). The DDP involves the *software interfaces* of the OS kernel, such as API calls from the kernel to the driver for initiating I/O transfers, and API calls from the driver to the kernel to manage resources and notify events (e.g., I/O completion). Moreover, the DDP involves the *hardware interfaces* of the storage controller, including memory-mapped I/O registers and DMA areas. Fig. 2 provides an overview of these interactions: when an application requests an I/O write operation, the OS invokes the device driver, and the device driver reads and writes the controller interface to start the I/O transfer. During the I/O transfer, the control flow is returned to the OS kernel, and the device driver is triggered again by the controller (e.g., by an interrupt request, IRQ) when the transfer has been completed.

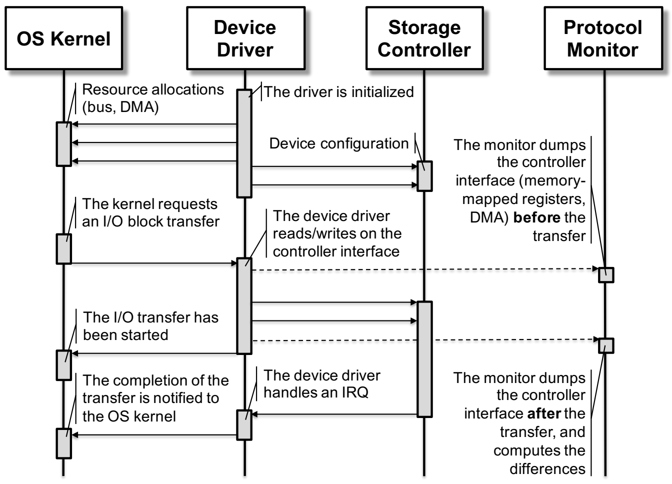

OS Kernel
Device Driver
Storage Controller
Protocol Monitor
Resource allocations (bus, DMA)
The driver is initialized
Device configuration
The monitor dumps the controller interface (memory-mapped registers, DMA) before the transfer
The kernel requests an I/O block transfer
The device driver reads/writes on the controller interface
The I/O transfer has been started
The completion of the transfer is notified to the OS kernel
The device driver handles an IRQ
The monitor dumps the controller interface after the transfer, and computes the differences

Fig. 2. Sequence of the interactions among the driver and the storage stack.

The several interactions between the layers of the storage stack, both in hardware and software (Fig. 1) exacerbate the complexity of the DDP. The hardware interactions include accesses to the interface of the storage controller, as prescribed by I/O standards such as the ones from IEC, ISO, and ANSI. These organizations have defined protocols that specify the types and roles of hardware registers, and the format and order of events, commands, and data to be exchanged with the storage controller. The software interactions include API calls from the OS to the device drivers and vice versa, by following the conventions required by API developers. Examples are API calls for allocating DMA buffers and for initiating an I/O transfer through the device driver.

The need for an automated approach to model the DDP is due to the lack of formal specifications for both hardware and software interactions. Typically, I/O standards leave out of scope, or do not strictly mandate, part of the specification of the storage controller, in order to let vendors fine-tune their products to stand out against their competitors (e.g., by providing multiple I/O channels to improve I/O performance, or by avoiding unessential features to reduce costs). Moreover, storage devices may deliberately differ from the standards, or even adopt proprietary specifications. The DDP also lacks formal specifications for the software interactions, as the documentation on API conventions is often lacking, inconsistent with the implementation, out of date, or only provided in natural language.

III. Proposed approach

The proposed monitoring approach has been designed with the following objectives:

- **Avoiding the need for the end-user to provide formal protocol specifications, or to know about device driver internals**: As discussed above, device drivers must follow a cross-layer protocol which is not documented with formal specifications. Moreover, requiring formal specifications from the end-users would make the approach not practical, since they lack in-depth knowledge about the drivers' internals and are not willing to invest efforts to scrutinize third-party protocols and source code for the sake of deploying a run-time monitor (as the use of a commodity OS, such as Linux, is to lower the development effort).
- **Low impact on performance**: The approach should have a negligible impact on the performance of the storage (e.g., in terms of throughput and latency); otherwise, users may not be allowed to deploy the approach in systems with high-performance requirements;
- **Coverage and accuracy**: To be useful and suitable for production environments, run-time monitoring should be able to point out misbehaviors of the device driver as soon as they occur, and should avoid false alarms that would needlessly trigger maintenance actions.

To fulfill these objectives, the proposed approach automatically learns the DDP from executions under failure-free conditions. The inferred DDP model is then deployed for monitoring the device driver and detecting protocol violations. The approach avoids the need for formal specifications or detailed knowledge of the device driver, by analyzing the behavior of the device driver as a *black box*, with focus on the *interfaces* between the device driver and, respectively, the storage controller (i.e., the contents of memory-mapped registers and DMA areas) and the OS kernel (i.e., API calls between the driver and the kernel). This approach is close to the idea of using behavioral models of sequences of API and system calls for intrusion detection [11], malware classification [12], and detection of regression bugs [13].

To keep low the performance overhead, and to achieve high coverage and accuracy, we designed the approach to only collect and analyze a small amount of selected information from the device driver. We carefully discarded information from the DDP model that exhibits high noise but contributes little in detecting protocol violations. For example, the monitor does not include in the DDP model the actual memory addresses of buffers with I/O transfers: these addresses are prone to unpredictable changes, since the OS memory allocator can allocate such buffers at different locations depending on non-deterministic factors such as the current memory availability; moreover, these addresses contribute little to the detection coverage of the monitor, since wrong addresses are likely to lead to memory access exceptions (that are already detected by the CPU) rather than to subtle protocol violations (e.g., omitted or incorrect protocol operations). Instead, we focus the DDP model on more stable information such as command opcodes: on the one hand, we can be confident that the device driver does not generate an invalid (e.g., nonexistent) command opcode under failure-free conditions; and, on the other hand, the occurrence of an invalid command opcode is a reliable indicator that the device driver is misbehaving. Thus, we hypothesize that monitoring a small, conservative subset of the device driver interface is useful to detect protocol violations and to achieve a negligible rate of false positives.

More insights on these problems will be discussed in the experimental part of this study (§ V).

In the first phase of our approach (called **model learning and synthesis**), we obtain traces of the execution of the device driver, by using probes at the interfaces of the device driver. The probes intercept the beginning and the end of I/O transfers, and record the state of the controller interface (see also Fig. 2). These traces are used to generate a monitor as a kernel module, which performs lightweight checks learned from the traces. In the second phase of our approach (**run-time monitoring**), we deploy the monitor inside the OS to check for violations of the I/O protocol.

The use of failure-free traces from the device driver is supported by the empirical observation that device drivers' failures are often *transient*: drivers behave correctly most of the time, and fail rarely due to the occurrence of subtle environmental conditions (such as timing of events due to multi-threading, interrupt handling, hardware events, and virtual memory) [14]–[19]. This transient nature allows us to use the device driver to train a reference model, and to use this model to detect failures of the driver itself (i.e., anomalous behaviors of the driver because of transient failures).

Even if drivers' failures are difficult to detect in production, the user of our approach can easily check that the training traces are failure-free, since the training runs are performed under *well-controlled conditions* (e.g., during planned downtime, or on a dedicated partition not used for storing actual data). In controlled conditions, it is possible to know in advance the expected inputs and outputs of the run. For example, when using synthetic load generators (e.g., tools such as *iozone* [20] benchmarking tool that we used in this study), the input data are created at run-time with a pseudo-random number generator, which are used to populate the storage or database; then, the data are read/written again and can be cross-checked using the same pseudo-random number generator as a reference. Moreover, in a controlled environment, the user can perform in-depth checks about the state of the storage at the end of each training run (for example, by using tools that check the consistency of the filesystem or the physical disk) [21]. These consistency checks cannot be used as run-time failure detectors for device drivers due to their overhead (e.g., the storage partition could not be accessed during the checks), thus they are performed infrequently in a production environment (e.g., periodically in terms of weeks or months, or when a system is rebooted). Instead, consistency checks can be freely performed in a controlled environment. Therefore, the failures are easy to detect once we can control and check the inputs/outputs of the runs and can perform thorough checks on the storage.

In the following subsections we present the full monitoring approach, by discussing the monitoring architecture (§ III-A) and the model learning techniques (§ III-B).

A. Architecture of the monitoring approach

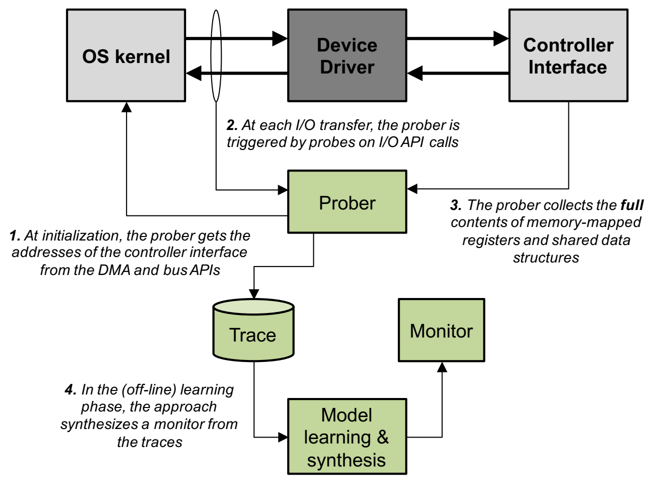
Fig. 3 shows the architecture of the proposed monitoring approach. The *prober* is triggered at every invocation of an I/O API call made by the OS kernel to the device driver. The *prober* inspects the controller interface, by reading commands, data, and other information stored in the registers of the controller and in DMA areas. The *prober* compares byte-to-byte these contents *before* and *after* the I/O API call, in order to identify and to record which parts of the controller interface (at byte granularity) have been modified by the device driver.

In the first phase (*model learning*, Fig. 3a), the *prober* dumps the whole controller interface into a trace; a *trace* consists of a sequence of such dumps. The trace is processed off-line in order to generate the *monitor* component. In the second phase (*run-time monitoring*, Fig. 3b), the controller interface is read and forwarded to the *monitor* to check the compliance to what is expected by the DDP model. A warning is raised in the case of violations. For example, if the driver issues incorrect commands (e.g., the commands are out of order, or do not follow the expected format), then a violation occurs.

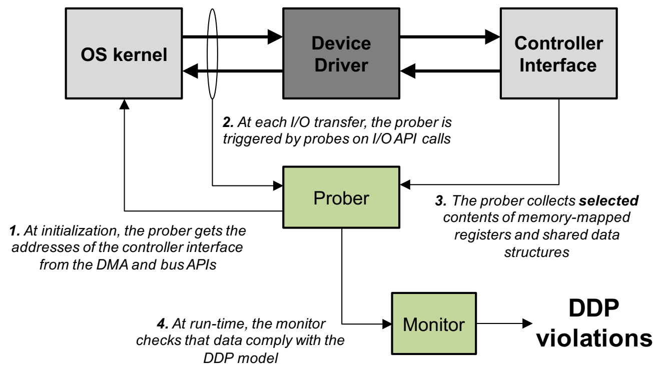
The two phases of the approach differ with respect to the *amount of data* that is read from the storage controller interface. In the first phase (*before deployment*, in which the performance cost of probing is not a concern, but only the correctness of the learned monitor), we collect full dumps of the controller interface. In the second phase (*in deployment*, in which performance is a concern), we only read selected parts of the controller interface, in order to minimize the performance overhead and to focus monitoring only on relevant parts of the interface. This is further discussed in the next subsection.

The *prober* component is triggered on *API invocations* from the OS kernel to the device driver, to track the changes made on the storage controller interface. We take advantage of *dynamic probing mechanisms* provided by most modern commodity OSes (such as *DTrace* for FreeBSD, Mac OS X and Solaris [22], *Kprobes* and *SystemTap* for Linux [23], [24], *VProbes* for VMware ESXi [25], and *Detours* for Microsoft Windows [26]). These mechanisms allow inserting *breakpoints* at run-time (in a similar way to a debugger) in the kernel. When the control flow triggers the breakpoint, a (customizable) *handler function* is invoked. This function can collect information from the kernel: for example, a breakpoint can be used to probe the invocations of an API function, where the handler can collect the input parameters and the return value of API calls, and any other information in the scope of the current call. The run-time overhead of these dynamic probing mechanisms is low enough to be applicable in many applications: for example, benchmarks from VMware reported that, in the worst cases, dynamic probes cause a throughput loss between 3% and 4.5% [25].

The *prober* acts on device driver APIs, which are functions that are visible to the upper layers of the OS and can be easily identified from documentation and OS development kits [27], [28]. One approach is to look for functions exported by the device driver for linking with the OS kernel: since device drivers in modern OSes are developed as pluggable modules (in order to be loaded on-demand, depending on the available hardware), the API functions of the device driver must be publicly exposed to other modules in order to allow linking. Alternatively, the device driver APIs can be identified by inspecting the header files or documentation of both the


OS kernel
Device Driver
Controller Interface
2. At each I/O transfer, the prober is triggered by probes on I/O API calls
Prober
3. The prober collects the full contents of memory-mapped registers and shared data structures
1. At initialization, the prober gets the addresses of the controller interface from the DMA and bus APIs
Trace
Monitor
4. In the (off-line) learning phase, the approach synthesizes a monitor from the traces
Model learning & synthesis

(a) Learning phase.


OS kernel
Device Driver
Controller Interface
2. At each I/O transfer, the prober is triggered by probes on I/O API calls
Prober
3. The prober collects selected contents of memory-mapped registers and shared data structures
1. At initialization, the prober gets the addresses of the controller interface from the DMA and bus APIs
Monitor
DDP violations
4. At run-time, the monitor checks that data comply with the DDP model

(b) Monitoring phase.

Fig. 3. Architecture of the proposed approach.

drivers and I/O subsystems, which are also public in order to ease the integration of third-party device drivers. In any of these cases, the *prober* installs breakpoints on these API functions in order to trace API calls. For more information and examples in the context of the Linux kernel, we refer the reader to the appendix A.

The approach focuses on persistently-mapped areas of the storage controller interface, such as memory-mapped registers and long-lived DMA areas, where the core part of the device driver protocol takes place, and does not consider transiently-mapped areas, such as temporary I/O buffers. The *prober* obtains the location of the memory-mapped I/O registers and DMA areas by probing or directly invoking kernel APIs (Fig. 2). The locations of memory-mapped registers and DMA areas are known to the OS, since their physical addresses need to be mapped to virtual memory addresses, and the device driver must map them using OS APIs; these API calls can be intercepted when the device driver is loaded and initialized. Moreover, memory-mapped registers can be identified using OS APIs for bus management; for example, PCI controllers use base address registers to configure the mappings, which can be read using OS APIs and tools such as “lspci” [29]. We remark that the approach does not require a different *prober* for each device, as the *prober* generically performs memory reads to take snapshots of the device interface. The only aspect that varies across devices is the base address and size of the memory areas to be read. These base addresses can be automatically obtained using OS APIs and tools.

Moreover, the *prober* takes care to handle the cases of special registers, such as registers that are write-only or that have side-effects on subsequent reads and writes. To deal with this kind of registers, we perform a preliminary analysis of the hardware interface, in which we exercise the storage and monitor one byte at a time of the hardware interface. If reading the byte raises an exception or causes side effects (leading to I/O failures), we configure the *prober* to avoid accessing that byte when applying the monitoring approach. It is important to note that omitting these registers has a negligible impact on the effectiveness of the approach, as these registers are relatively rare, and since a fault that affects such a register would also indirectly affect the contents of other status registers. In the case of registers that can exhibit different behaviors (such as, due to register banking or, more often, registers that are influenced by the device configuration at initialization time), the model learned by our approach is tailored to the behavior that was observed at training time. If, for any reason, the end-user needs to change the configuration of the device driver, then the model should be re-learned in order to take into account the new behavior.

The monitoring begins when a *mount* operation is started on a storage partition, and lasts until an *unmount* operation of the storage partition completes. Thus, the monitoring includes all accesses to the device interface that happen during the mount and unmount operations. The monitoring does not include the period when the OS loads the device driver before mounting of the storage partition (e.g., the time during which the hardware controller is polled, and part of the device driver data are initialized), since in these periods no I/O operation can be performed and data corruptions are highly unlikely. For the same reason, the monitoring does not include the period between between the unmount of the partition and the unload of the device driver (e.g., in which the hardware device is powered off). In the case that a fault hits the device driver before mounting the partition, the monitoring approach is still able to detect protocol violations caused by postponed effects of the fault that may surface later during I/O operations.

B. Monitor learning

The approach uses execution traces from the *prober* to generate a monitor, to be deployed at run-time (Fig. 3). The monitor embeds a state model of the device driver protocol, where the *states* represent snapshots of the contents of the controller interface (i.e., the samples in the trace), with one state for each unique snapshot of the controller interface; and *transitions* connect two states if they appear consecutively at least one time in the learning trace. Before generating the state model, the learning technique truncates and transforms parts of the controller interface in the trace (i.e., states are projections of the contents of the controller interface) in order to improve accuracy. More specifically, the learning technique performs the following four steps to process the trace.

▹ ***1. Execute Device Driver with Full Tracing***. We execute and trace the device driver by collecting "full" traces of the controller interface (e.g., the whole contents of memory-mapped I/O registers). The raw trace consists of a sequence of dumps of the changes made on the controller interface. A sample provides the new values (represented as a vector of bytes) written on the controller interface during the invocation of the probed API. In the simple example of Fig. 4, the three samples show new values that are written in the first byte of the interface, and from the third byte onward.

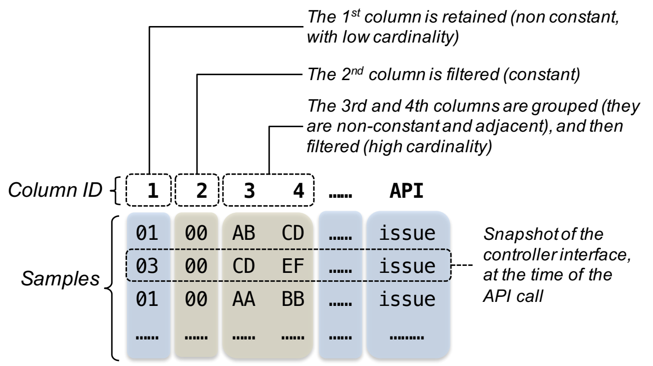

The 1st column is retained (non constant, with low cardinality)
The 2nd column is filtered (constant)
The 3rd and 4th columns are grouped (they are non-constant and adjacent), and then filtered (high cardinality)
Column ID
1
2
3
4
……
API
Samples
01 00 AB CD …… issue
03 00 CD EF …… issue
01 00 AA BB …… issue
Snapshot of the controller interface, at the time of the API call

Fig. 4. A simple example of trace filtering.

During this first tracing step, we dump the whole controller interface. From this full trace, the approach learns which parts of the interface are useful for monitoring, by identifying the type of information that is stored in the bytes of the controller interface (such as bitfields, commands, or addresses). A subset of these bytes is selected according to their type. This approach is motivated by the observation that most of the information in the controller interface is not useful for detecting protocol violations. In particular, our approach avoids collecting parts of the controller interface that are highly-variable and noisy, but that do not reflect the DDP (for example, the specific memory address used for an I/O buffer), and that may increase the monitoring overhead and lead to false alarms (e.g., addresses may vary across executions in non-deterministic ways).

▹ ***2. Filter Columns from the Full Trace***. This step analyzes the raw traces to identify the relevant parts of the controller interface to be monitored. The samples are divided into *bytes*; a *column* of the trace is the set of bytes at the same position of the samples (e.g., the second byte of each sample). This step generates a list of the subset of columns that should be traced by the monitor. For example, in Fig. 4, we remove the second and third columns, and retain the first one for further analysis. Our approach identifies the following cases.

Constants: We analyze individual columns, and check which ones always exhibit the same value across the sample, removing columns with only constant values (i.e., with single cardinality). For example, the monitor does not need to collect and check registers that are only modified at start-up (e.g., configuration registers) but are constant during I/O operations.

Large fields: After filtering columns that are constant, we identify and group columns at adjacent positions in the residual trace. Typically, such adjacent columns represent a multi-byte register of the device interface. Then, we apply on the group as a whole the next filters for *bit-fields* and *addresses*. For example, in Fig. 4, we initially filter the column 2, since it is constant; then, we group the two adjacent residual columns 3 and 4. Later in the process (see the discussion on *addresses*), the group represented by 3 and 4 will be filtered as a whole.

Besides checking that columns are adjacent, we perform an additional check that the columns are not unrelated and adjacent by chance (for example, a column that represents a status can be followed by a column with a command). We avoid to group adjacent, unrelated columns by checking whether there is *at least one sample in the trace where all the bytes in the group vary at the same time*. For example, in Fig. 4, the group with the third and fourth columns satisfy this condition, since all bytes in the group vary from one row to the next one (e.g., from *ABCD* to *CDEF*, and from *CDEF* to *AABB*). We can expect that multi-byte groups that never vary together over a very long trace can be reliably considered unrelated; for example, it is very unlikely that an address always varies with respect to only one byte at a time.

Bit-fields: We check whether columns contain bit-fields, such as, registers that represent the state of a group of I/O channels, with one bit per channel. We look for differences between the value of a sample at a given column, and the value at the same column of the immediately-previous sample. We perform this comparison for every pair of consecutive samples in the trace. If there are differences, and the differences involve only one bit in most of the cases (e.g., more than 90%), then it is likely that the column represents a bit-field. The threshold takes into account the possibility of rare cases in which two or more bits are accidentally changed at the same time (e.g., due to concurrent I/O requests). We rewrite these columns by replacing them with the *count-of-modified-bits* that are changed during the driver API call. For example, this count may be 0 (no bit is changed), 1 (only one bit changed), or greater than 1 (we allow for sporadic variations of more than one bit); the value can be either positive or negative negative depending on whether the bits are set or cleared, as these operations can have different meanings for the device driver protocol. This simplification allows to keep the monitor small and robust (as we do not want to over-train the monitor to look for the exact position of the bits that are set/reset, since it would make it more prone to false positives), and still retains useful information for monitoring purposes.

Addresses: We remove from the monitor the columns that are noisy and highly-variable, by checking for columns with a *high number of different values* across samples. This approach leverages the fact that protocol commands and status information are likely to have a small cardinality (e.g., few tens of possible values), while noisy information (such as memory addresses) tend to vary wildly across executions. Even if potentially invalid addresses are not checked by the monitoring approach, the hardware can still identify and notify them. Invalid pointers used for I/O (such as, pointers to temporary I/O buffers that are uninitialized, or used after they are freed) are translated and checked by the I/O and CPU memory management units (MMUs), respectively when the

I/O controller and the device driver access to them. In turn, the MMU will raise an exception on invalid pointers, since there is no entry for them in the address translation tables; moreover, the storage controller can write an error code in a status register, and raise an interrupt to denote the failure. The proposed approach will monitor such status registers since they have a small cardinality.

▹ ***3. Execute Device Driver with Partial Tracing***. Once relevant columns have been selected in the previous step, the device driver is executed for a second time. In this case, we only collect "partial" traces of the storage interface (i.e., only columns selected in the previous step). This additional run of the device driver is needed to get execution traces that are free from perturbations that might have been caused by the "full" tracing, since reading the whole contents of the controller interface has a high overhead and can perturb the timing of events in the trace (the notorious "probe effect" [30]). In the "full" trace, we only need to identify the type of information contained in the columns (such as bitmasks, commands, addresses), and thus we are not concerned with the relative ordering of the events; instead, the "partial" tracing allows to get a better profile of the driver's behavior, as the relative ordering of events and commands is better preserved. The partial trace converts the columns in the same way of the previous step (i.e., converting them in bit-field counts and coalescing them).

▹ ***4. Synthesize Monitor from the Partial Trace***. We obtain a monitor for the DDP from the filtered trace of the previous step. We generate one distinct monitor for each memory-mapped region of the storage controller, as these regions can evolve independently from each other: for example, a storage controller may provide multiple I/O channels to improve performance, by exposing different groups of memory-mapped registers.

In the case of large DMA areas, which are often organized as large arrays (e.g., where each element represents a different I/O port or I/O operation), we split the area into individual elements of the same size, and generate one distinct monitor for each of them. The size of the elements is easily determined by looking at the definitions of data structures that are used by the device driver to interpret the raw contents of the DMA area. These data structures can be defined in a public header of the OS, in the case that the data structure is part of a generic standard that is not specific for an individual device driver, such as the PCI standard. Once the element size is known, the columns in the traces are partitioned in different traces according to the element they belong to. If the definition is not public (e.g., for closed-source drivers), the user of our approach would need to determine the size of the data structure from the executable binary of the device driver, by analyzing accesses to arrays in DMA areas using reverse engineering techniques and tools [31]–[33]. As we do not analyze binary drivers, we leave this topic out of the scope of this work, and assume that the element size is known.

The monitor maintains and checks at run-time a global set of *allowed states*. The distinct vectors of bytes in the (sub-)trace are turned into allowed states of the protocol. For example, in the simple case of Fig. 4 with only one column, we generate two allowed states from the two distinct values of the first column. Section V-B provides in Fig. 7 another example in the context of a real device driver. When the *monitor* component is synthesized from the filtered trace, the set of distinct allowed states is stored in a hash table, which is queried at run-time. At each API invocation of the device driver at run-time, the monitor collects the selected bytes from the controller interface, filters the columns (by only keeping the columns selected in the second step), and checks that the occurred state is within the set of the allowed ones. Otherwise, if the monitor detects that the current state is not compliant to the model, it raises an alarm to notify the DDP violation; in turn, the alarm can be reported in system logs to ease maintenance, or it can be fed to an automated system to initiate a recovery process. These uses of the monitor are further discussed in § VI-A.

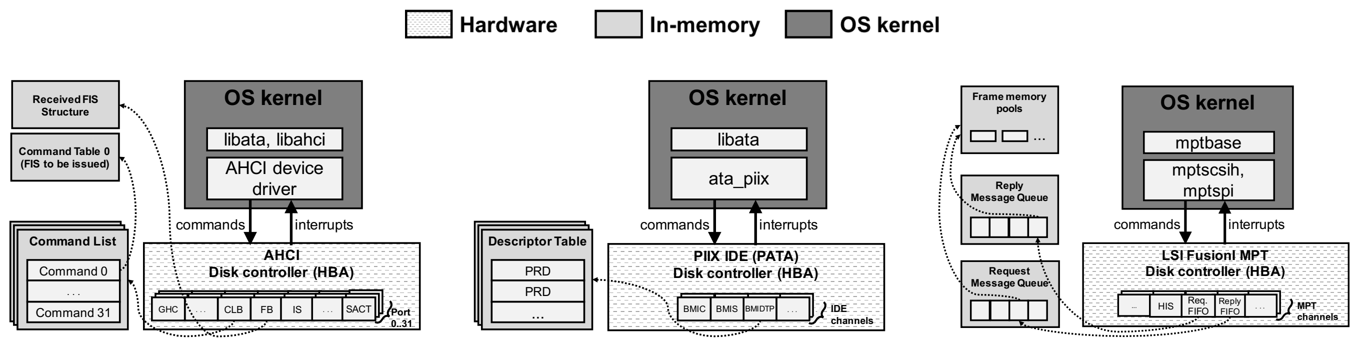
IV. Case study

In this paper, we evaluate the proposed monitoring approach on the following three storage device drivers from the Linux kernel: (i) SATA/AHCI (i.e., the *ahci* driver); (ii) Intel SATA/PATA (i.e., the *ata_piix* driver); and (iii) LSI Fusion MPT (i.e., the *mptspi* driver). We selected these device drivers since hypervisors (such as VMware's products) provide full hardware virtualization support to run them, which enables us to perform fault injection experiments in a controlled way, as discussed in § V. Moreover, they are a relevant target since Linux is largely used for business-critical servers, which over the years has matured to support a broad range of storage devices, protocols and filesystems [34], [35]. Furthermore, device drivers from the Linux kernel have also been ported to other systems, such as the VMware ESXi hypervisor [36].

These drivers are representative and complex case studies, as they involve all the subtleties that are typical of real-world storage device drivers. SATA/AHCI is a modern I/O technology (the latest revision dates back to 2014) that is currently used by a large number of controllers on the market (e.g., controllers from Intel [37]). LSI Fusion controller technology is commercialized and widespread across enterprises [38], and it is representative of the more general category of SCSI controllers (as the LSI Fusion driver is included in the architecture of the SCSI subsystem of the Linux kernel). Finally, we included the SATA/PATA driver even if it is based on an older technology, since it is representative of legacy device drivers in which the approach could be applied by some end-users. Moreover, the SATA/PATA case study is complementary to the other two drivers, which are more complex, e.g., in terms of size of the device interface, and it allows us to evaluate how well the proposed approach can perform in a relatively favorable case. Fig. 5 shows the architecture of these three case studies. Further information on these drivers, and on the application of the monitoring approach, is provided in the appendix.

V. Experimental evaluation

In the following, we first introduce the experimental setup, and we provide information about how the monitoring approach has been trained for the three target device drivers.



(a) SATA/AHCI. (b) Intel SATA/PATA. (c) LSI Fusion MPT.

Fig. 5. High-level architecture of the device drivers analyzed in this study.

Then, we evaluate the proposed monitoring approach in terms of coverage, accuracy, and performance impact.

A. *Experimental setup*

The *System Under Test* (SUT) that we considered for the experiments is a Linux machine running a Fedora 21 distribution, and the Linux kernel version 3.19. We performed both fault injection tests and long-running stress tests on the SUT, to evaluate on the one hand the coverage of protocol violation (by deliberately forcing violations through injected faults), and on the other hand the accuracy and overhead of monitoring (by measuring false positives and performance under failure-free conditions).

In order to inject faults in the SUT, we execute it within a *virtual machine* (VM), while we orchestrate the experiments from the host machine on which the SUT resides. Virtualization is extensively used in fault injection experiments in OSes [39]–[41], since it is useful to prevent the injected faults from propagating from the SUT to the orchestration software. Indeed, using a VM, the injected faults remain isolated within the SUT, and the orchestration software can correctly save the data from the current experiment, and to start the next one. We adopt the VMware ESXi 6.0 hypervisor to run the SUT within a VM, which fully emulates the disk controllers to be managed by the three target device drivers (i.e., the disk controller of the VM appears like a physical disk controller to the device driver in the VM) [42]. The VM runs on a DELL workstation with an 8-core Intel Xeon 1.80Ghz CPU, with 64GB RAM, and with a PERC H730 Mini HDD controller. The VM was configured with a 4-core 1.80Ghz virtual CPU and 4 GB RAM. Fig. 6 shows the experimental setup and workflow of fault injection.

We remark that the virtual devices used for the evaluation are not simplified versions of their real counterparts, but provide a full software implementation of storage I/O interfaces: this property allows the VM to execute unmodified and legacy device drivers of commodity OSes, such as Windows, Linux, and even the VMware ESXi hypervisor running on the virtual devices using nested virtualization [42]–[44]. Therefore, the virtual devices expose a controller interface that, from the point of view of the OS and of the device drivers, is fully equivalent to real devices, and that represents a realistic target for evaluation of the proposed monitoring approach. Using

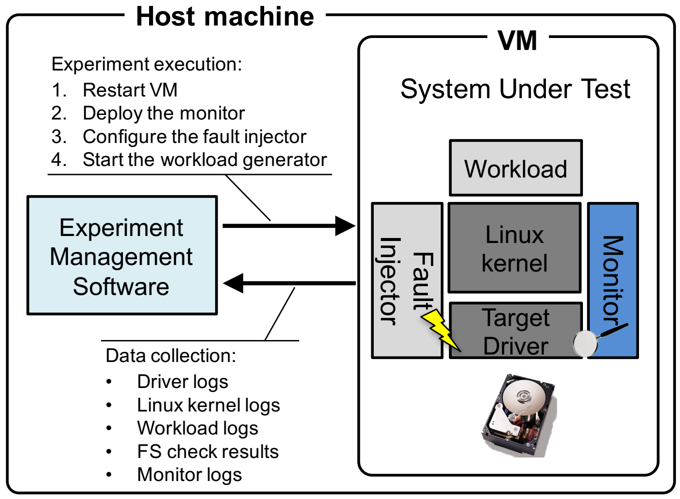


Fig. 6. Fault injection setup.

virtual devices brings significant benefits to the experimental evaluation, since a large number of fault injection experiments can be fully automated in a reliable way (e.g., by assuring the correct collection of error logs from the OS), and could be accelerated by leveraging parallelization and by using VM snapshots to quickly reboot VMs across experiments. Moreover, the storage stack of the VMware ESXi hypervisor (in particular, the VMFS filesystem) adopts a lightweight virtual-to-physical mapping in order to achieve high-performance, concurrent read/write access on the underlying disk, thus incurring in minimal overhead [45].

In our experiments, we exercise the I/O storage stack by using three well-known I/O-bound workloads [46], [47]:

- **IOzone**: A benchmarking tool for filesystems [20]. IOzone generates a mix of file operations, using different read and write patterns (e.g., random, sequential, fread, fwrite, strided, repeated), different sizes (e.g., by changing both 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 variety of data- and metadata-intensive operations on a pool of random text files [47]. Postmark creates the pool of files with uniformly dis-

tributed sizes, and performs a sequence (namely *transaction*) of random I/O operations (e.g., file creation, deletion, read, and append);

- **SQLite**: This benchmark program comes from the Phoronix open-source test suite [48]. The program exercises SQLite by performing a mix of SQL queries to store and to retrieve tuples.

The *IOzone* workload is not meant to be representative of a specific user application, but it is instead a *microbenchmark* used to evaluate peak performance, by generating high-volumes of specific I/O operations. In contrast, the *Postmark* and *SQLite* workloads *macrobenchmarks*, which perform a mixture of multiple file system operations, and aim to simulate more realistic I/O access patterns. These benchmarks are derived from server and database applications, and focus on the I/O request patterns and sizes that most frequently happen in these kinds of applications.

In our experiments, we use the *IOzone* workload to train the monitoring approach. This choice reflects the likely use case for the proposed monitoring approach: the user wants to train the monitor in his own testing environment, but cannot exercise the storage stack with a workload representative of the actual workload in production (e.g., the user may lack historical data, or the production workload is too complex to be replicated in a testing environment). In such case, a synthetic workload generator would be a more practical solution. Therefore, we evaluate how our monitoring approach would fare when the monitor is exposed to a workload different than the training workload: for this reason, we evaluate the monitor by training it with *IOzone*, and testing it both with *IOzone* (an optimistic case where the training workload matches the operational workload), and with the *Postmark* and *SQLite* workloads (a pessimistic case where the training workload is not representative of the operational workload).

B. Impact of filtering and learning duration

By applying the learning technique on the three device drivers, we can evaluate how the filtering of columns from the trace can actually reduce the information that is collected from the device drivers. We report in Table I the extent of the controller interfaces, respectively before filtering (i.e., the unaltered controller interface) and after filtering (i.e., the number of bytes that are retained after removing columns that do not contribute to monitoring, such as constants and addresses) as discussed in § III-B. These data show that the reduction of the monitoring interface is quite significant, as it ranges between 73.53% (the Intel SATA/PATA driver) and 99.41% (the SATA/AHCI driver).

To better understand the information selected by the filtering technique, we look in more detail at the SATA/AHCI, for which the technique selected 3 registers to be monitored (see the appendix A for more information): the Port and Host Interrupt registers, and the SACT register. The values read from the SACT register are processed as a bitmask, and are converted into a count of the bits that vary between the beginning and the end of the API call of the device driver (a positive or negative number, depending on whether the bits are set to 0 or 1). The SACT register is used by the driver to specify the entry of the command list in which a command has been inserted; the other two registers are used by the controller to point out the completion of commands. The remaining registers that were filtered out are either constants (such as, the base address of the command list) or protocol-independent values, such as the memory address and contents of the I/O transfers. Fig. 7 shows a finite state machine representation of a subset of the sequences (for better readability) observed for these three registers. Once the monitor focuses on selected parts of the controller interface, the possible sequences of values for these registers follow few and repetitive paths.

Selecting a small amount of information keeps the monitoring overhead low (since less data needs to be read and inspected, as further discussed in § V-E), and avoids false positives, since we do not consider data that have a high variability and do not contribute to identifying protocol violations. Later in the paper, we evaluate whether such simple information is useful to detect protocol violations of faulty device drivers.

To evaluate the influence of the workload on the training of

Table I
EXTENT OF THE MONITORED CONTROLLER INTERFACE WITH AND WITHOUT FILTERING.

Device Driver	**Extent of regions (bytes)**	
	Before filtering	After filtering
LSI Fusion MPT	1,312	44 (-96.65%)
SATA/AHCI	512	3 (-99.41%)
Intel SATA/PATA	34	9 (-73.53%)

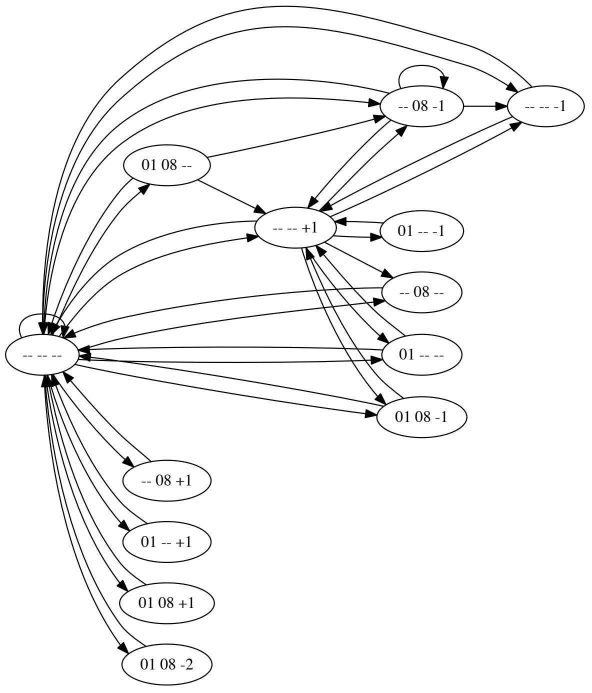


Fig. 7. A subset of event sequences in the monitor for SATA/AHCI.

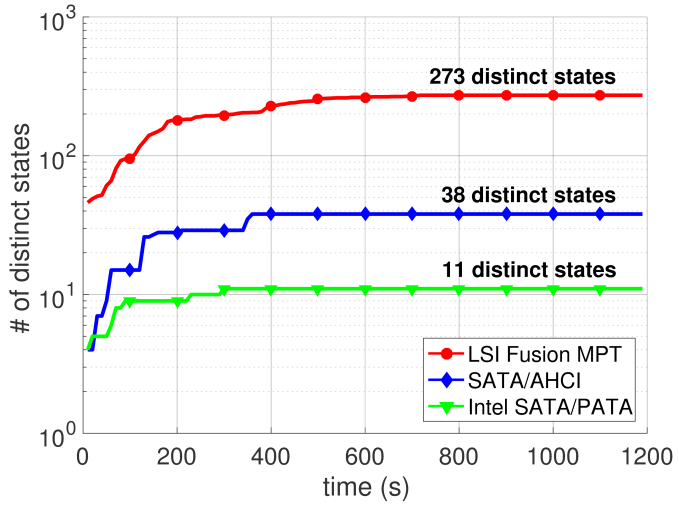


Fig. 8. Number of distinct states with respect to the duration of the training workload for the three target drivers.

the monitor, we analyze how much training data are needed to generate the model. For this purpose, we perform an additional experiment in which we evaluate the number of *distinct* states that occur as the training workload keeps executing. To this goal, we continuously repeat the execution of IOzone up to one hour, and collect a long trace of samples; then, we compute the cumulative number of distinct states in the trace over time, by applying the filtering to increasing subsets of the trace. Fig. 8 shows the outcome of this experiment on the three device drivers. Most of the distinct states occur at the beginning of the experiment, and the occurrence of new distinct states gradually slows down over the course of the experiment. Some of the new states occur in the middle of the experiment due to occurrence of relatively-rare combinations of events (e.g., two or more command entries that are concurrently being used). After few minutes, the number of distinct states becomes stable, as no new information is observed even if the workload duration is extended. The number of distinct states saturates after 5 minutes of execution in the best case (Intel SATA/PATA driver), and after 11 minutes in the worst case (LSI Fusion MPT driver). In all cases, a few minutes of execution sufficed to get a sufficient number of samples to train the model, as further training data do not extend the monitor. Thus, we conclude that it is feasible to train the monitor without performing long-running tests, which makes the approach quickly deployable in practice. The actual coverage and accuracy of the monitor derived in this way are validated in the next subsections.

C. Evaluation of coverage

In this section, we evaluate the proposed approach in the presence of faults in the device drivers. When we inject faults, three possible outcomes can occur:

- **Crash** or **stall** of either the OS or the workload. These cases are identified by checking crash logs in the SUT (e.g., kernel panic messages), and by testing if the SUT is still responsive when the experiment ends.
- **Data corruption**, that is, an execution that neither crashes nor stalls, but data are corrupted at the end of the experiment. We identify such cases by (i) checking if the workload produces output files comparable to the fault-free execution, and (ii) checking if the filesystem data structure are inconsistent (e.g., damaged directory trees and "orphaned" files) by leveraging the *fsck* filesystem verification tool;
- **Correct execution** of the SUT. This outcome occurs when the injected faults do not generate any error in the SUT (for example, when corrupted data are not accessed or are over-written with correct data).

In particular, our analysis focused on *data corruption* failures. Such failures are subtle and not easily detected by checking for crash messages or heartbeats. We quantify the *coverage* of our proposed monitor, which is *the percentage of experiments with "data corruption" failures, and the corruption is detected by the monitor*.

We inject faults in the device drivers during the execution of the workload. We leverage the fault injector developed by Ng and Chen for evaluating fault tolerance strategies for OSes [49]. The fault injector emulates common patterns of software bugs, based on empirical studies on OS faults [50], [51]. The injector emulates the following fault patterns: *Assignment*, i.e., incorrect source or destination in assignment instruction; *Control*, i.e., incorrect logical condition in loop or branch; *Parameter*, i.e., incorrect parameter in a function call; *Pointer*, i.e., an incorrect memory pointer computation; *Omission*, i.e., an omitted instruction. These kinds of bugs are emulated by removing or modifying the existing code in the device driver. Faults are injected by modifying the binary code of the driver, where the original instructions are replaced with faulty ones [52], [53] (e.g., by replacing instructions with *no-ops* to emulate omissions). We provide further background on fault injection in Section VI-B.

These patterns of software bugs can lead to protocol violations if they are injected in the context of device drivers' code. To understand the relationship between fault injection and protocol violations, we consider the four categories of protocol violation bugs that were identified by Ryzhyk et al. [10], [54]. Quoting these studies:

- **Timing Faults**: "Device state transitions can be triggered by the passage of real time. The driver simulates such transitions using timeouts. Forgetting to put a timeout statement in the appropriate place or using an incorrect timeout value is likely to lead to failure of subsequent commands issued to the device."
- **Value Defects**: "The driver and the device exchange data, including device descriptors, configuration commands, and I/O transfer descriptors, via memory and registers. Defects related to handling of device data include endianness errors, incorrect use of register bit fields, sending invalid data values to the device, and incorrectly interpreting values received from the device."
- **Ordering Defects**: "In order to correctly control the device, the driver must keep track of the internal state of the device state machine. Errors occur when the pro-

grammer’s mental model of this state machine diverges from its actual implementation. These errors may lead to the driver issuing a sequence of commands to the device that fails to meet its intended goal or even leaves the device in an invalid state.”
- **Data Races**: “The device and the driver may engage in shared-memory communication using DMA and memory-mapped I/O. Access to shared memory regions is synchronised using interrupts, device registers, and memory barriers. Incorrect use of synchronisation can lead to a race between the driver and the device.”

```
cmd &= ~(PORT_CMD_ASP | PORT_CMD_ALPE);
cmd |= PORT_CMD_ICC_ACTIVE;

writel(cmd, port_mmio + PORT_CMD);
readl(port_mmio + PORT_CMD);

/* wait 10ms to be sure we've come out of LPM state */
ata_msleep(ap, 10);
```

(a) Timing fault (libahci.c:712).

```
/*
 * Setting up proper handlers for scatter gather handling
 */
if (pdev->device == MPI_MANUFACTPAGE_DEVID_SAS1078)
  ioc->add_sge = &mpt_add_sge_64bit_1078;
else
  ioc->add_sge = &mpt_add_sge_64bit;
```

(b) Value defect (mptbase.c:1812).

```
/*
 *  if we get a data underrun indication, yet no data was
 *  transferred and the SCSI status indicates that the
 *  command was never started, change the data underrun
 *  to success
 */
if (status == MPI_IOCSTATUS_SCSI_DATA_UNDERRUN &&
    xfer_cnt == 0 &&
    (scsi_status == MPI_SCSI_STATUS_BUSY ||
     scsi_status == MPI_SCSI_STATUS_RESERVATION_CONFLICT ||
     scsi_status == MPI_SCSI_STATUS_TASK_SET_FULL)) {
        status = MPI_IOCSTATUS_SUCCESS;
}
```

(c) Ordering defect (mptscsih.c:674).

```
spin_lock_irqsave(&ioc->scsi_lookup_lock, flags);
scmd = ioc->ScsiLookup[i];
spin_unlock_irqrestore(&ioc->scsi_lookup_lock, flags);
```

(d) Data race (mptscsih.c:2471).

Fig. 9. Examples of drivers’ code where the fault injection approach generates protocol violation bugs [54].

When faults (e.g., assignment, control, etc.) are injected in drivers’ code, they can generate protocol violation bugs of these four categories (e.g., the injected fault leads to incorrect data exchanged with the device). Fig. 9 provides examples of drivers’ code and injected faults, taken from the device drivers used in our study. Since this kind of code is widespread in device drivers, fault injection can generate a large number of protocol violation bugs for evaluating the coverage of the proposed approach. The examples include:

- In the context of AHCI (e.g., Fig. 9a), timing primitives are used for Link Power Management (e.g., to wait for link to become ready, or for the completion of a hardware reset, before submitting more commands). However, the delay embedded in the primitives’ parameters has to be properly tuned (e.g., the AHCI module was revised to configure the default delay for the specific controller flavor [55]). When applying *omission* or *parameter* faults on these primitives, the injected fault results in a missing or incorrect wait. The monitoring approach is intended to detect out-of-order writes on the storage interface resulting from bad timing; for example, a bug in the Adaptec AAC-RAID device driver [56] updated the phase field in the SCSI command scratchpad area after that the command had already been completed and the area has been re-allocated, causing an inconsistent phase value written on another command.
- In order to support controller flavors that subtly deviate from the standard, device drivers perform special operations as workarounds. For example, the LSI Fusion MPT (Fig. 9b) provides a workaround for the LSI 1078 chipset on systems with more than 36GB of memory [57]. Injecting a *control* fault emulates the lack of such workaround and the resulting misread/write of the device interface. A regression bug of this kind occurred in libATA [58], which issued an incorrect mode sense command, which could have been detected using a monitoring module trained before the regression.
- The LSI Fusion MPT (Fig. 9c) tracks the status of I/O transfers using a state machine, which is updated on interrupts from the device controller. This state machine includes transitions to handle several corner cases [59]; the example handles the case of an “underrun” indication when no data has been yet transferred. Injecting a *control* or *assignment* fault on this conditional construct emulates a lacking or mistaken transition in the state machine. A similar bug affected the JMicron JM20337 driver [60], which failed because of an anomalous “Check Condition” status register not handled by the driver (but detectable by monitoring the controller interface), that happens when the Force Unit Access (FUA) mode is disabled.
- The LSI Fusion MPT (Fig. 9d) uses an array of shared variables to hold concurrent SCSI commands, which need to be guarded against interrupts handlers triggered by the device. Injecting *omission*, *assignment*, or *pointer* faults on accesses to such shared variables corrupts values read or written on the variable (as would happen in the case that the access had not been protected from *data races* by using a lock and temporarily disabling interrupts [61]). For example, a bug-fix in the qla2xxx driver [62] anticipated a mutex lock to protect against inconsistent reads of the controller state, which caused the overlap of several I/O writes on the Qlogic controller interface.

We injected faults in the most frequently called functions of the drivers, which we found using OProfile during the execution of the three workloads [63]. The fault injection targets include functions that populate DMA areas with I/O frames (*ahci_qc_prep*, *ata_bmdma_qc_prep*), issue the I/O commands (e.g., *ahci_qc_issue*, *ata_bmdma_qc_issue*, *mptscsih_qcmd*), handle interrupts (e.g., *ata_bmdma_interrupt*, *mpt_interrupt*), and configure and check the status of the I/O controller (e.g., *mpt_config*, *ata_bmdma_setup*). Overall, we performed 210, 2160 and 3600 fault injections on distinct code locations, respectively for the SATA/AHCI, Intel SATA/PATA, and LSI Fusion MPT drivers, where the number of injected faults depends on the size of the drivers' code and on the number of fault locations that we found in the driver. At each experiment, we randomly choose the fault type and location within the target functions to be injected.

From the results, we determined the outcome of the injected fault as mentioned before and, for data corruption failures, the outcome of failure detection. Fig. 10 and Fig. 11 show respectively the percentages of the failure modes, and the coverage of the monitor. In many cases, the experiments ended with no failures, in which the injected fault does not affect the device driver (i.e., the corrupted data were ignored or overwritten, or the corrupted instructions did not alter the behavior of the driver). This phenomenon is commonly observed in fault injection experiments [64]. Nonetheless, a non-negligible number of experiments led to data corruption failures, by affecting workload data (causing incorrect outputs) and/or the filesystem state (such as, orphaned inodes, invalid partition information, or corrupted superblocks).

The monitor detected data corruptions in 87%, 88%, and 76% of cases respectively for SATA/AHCI, Intel SATA/PATA, and LSI Fusion MPT (Fig. 11). In these experiments, the injected faults caused the device driver to perform erroneous writes on the device interface, which violate the reference model of the monitor. In the remaining (undetected) cases, when the kernel invokes the APIs of the device driver, we observed that the injected faults caused the driver to prematurely return the control flow to the kernel, without accessing to the hardware interface. In these cases, the model stays in the current state (as the contents of the device interface have not been changed); the current state is considered a correct one by the model, but the driver is failing since it was supposed to change the contents of the device interface. The undetected corruptions can be attributed to the relative simplicity of the learned model, which does not consider detailed information about the calls to the driver API (e.g., the input and output parameters of the calls), that would increase the detection coverage, but would also make the monitor more prone to false alarms. These failures are more easily detected by checks performed by the kernel, e.g., by handling error codes to detect the premature termination of the driver API calls (see also the discussion about error checks in the Linux kernel, at the end of this subsection). Thus, despite these undetected cases, the monitor can achieve a good trade-off in terms of coverage and accuracy (see also § V-D).

It is worth noting that the corruption detection coverage is high even when the workload used for training the monitor differs from the workload of the experiments (i.e., in Fig. 11, the coverage for the Postmark and SQLite workloads is comparable to the coverage for the IOzone workload). This behavior eases the adoption of the proposed approach, even without training it with a workload representative of the production environment, since training with a generic synthetic workload generator (e.g., *IOzone*) suffices to achieve a high coverage. Such results can be explained by the fact that the architecture of the storage stack (Fig. 1) involves several layers between user applications and the device driver. We hypothesize that the variations of I/O access patterns of the workloads (e.g., the frequency and size of I/O reads and writes) are not perceived by the device driver, as the intermediate layers (filesystem, and I/O scheduling, buffering, and caching) turn them into a regular sequence of I/O interactions. We will further analyze this behavior in another experiment presented in the next subsection.

We also evaluated how the proposed monitor complements the existing error checks inside the I/O storage stack. For example, I/O standards (including the ones implemented by the three target drivers of this study) provide mechanisms to detect and to handle I/O errors (mainly, errors that arise from physical faults of disks and interconnections), by raising special interrupts and error codes that can be handled by the OS. In turn, the OS reports I/O error messages (e.g., through the system's logs or management dashboards) to the user to notify the need for maintenance. However, these mechanisms are typically not designed to detect misuses of the controller interface by buggy drivers; therefore, it is worth to consider that our monitoring approach is deployed alongside these error checks to further improve the overall fault detection coverage. Thus, we evaluated the overlap between (i) the set of experiments with corruptions that were detected by the monitor; and (ii) the set of experiments with corruptions that were detected by error checks inside the I/O storage stack, by looking for error messages in the logs of the Linux kernel.

In all workloads and target drivers, we found that there is always a part of the corruptions not detected by the error checks of the Linux kernel, but only by our monitor (26% of all corruptions, which is a subset of the covered cases in Fig. 11). A minor part of the corruptions (13%) were only detected by the Linux kernel, which are the cases where the fault causes the premature termination of a call to the driver API; the remaining part of corruptions (56%, which is another subset of covered cases in Fig. 11) were detected both by the kernel and the monitor. In particular, the kernel was not able to detect faults that neither affected the control flow nor triggered exceptions from the controller, but that caused omitted or wrong writes on the controller interface. Therefore, the monitoring approach complements the kernel by covering faults that would be otherwise unnoticed, thus improving the overall reliability of the storage stack.

D. *Evaluation of accuracy*

We investigate the tolerance of the proposed approach against false alarms, by running the SUT with the monitor for a long time, without fault injection. As in the previous experiments, we train the monitor using *IOzone*, and deploy this monitor in the SUT. We then stress the three device drivers for a week using *IOzone*, *Postmark* and *SQLite*. We repeatedly run a workload by re-launching it right after an execution has been completed, and we alternate between the

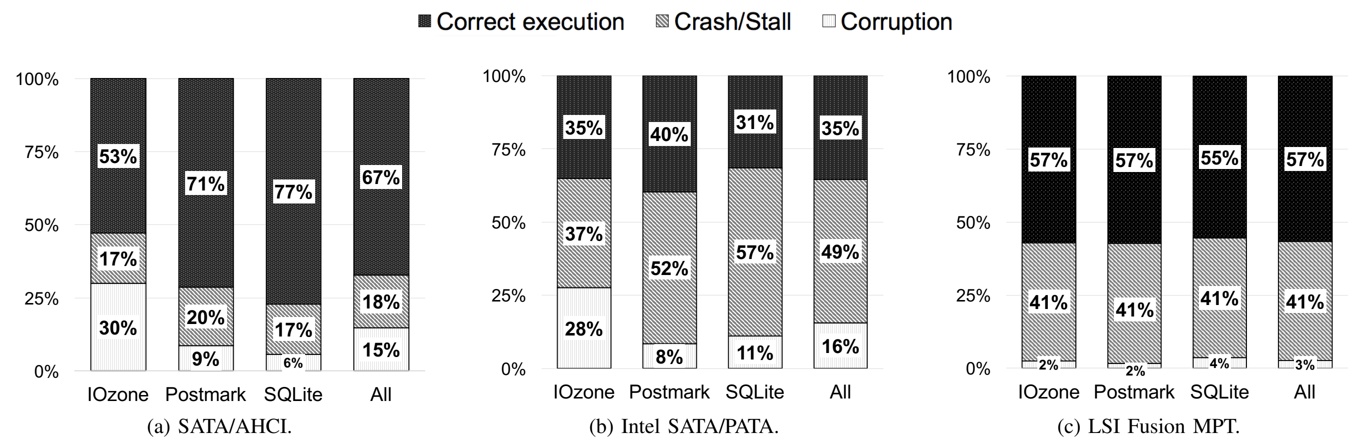


Fig. 10. Fault injection outcomes for the three target device drivers.

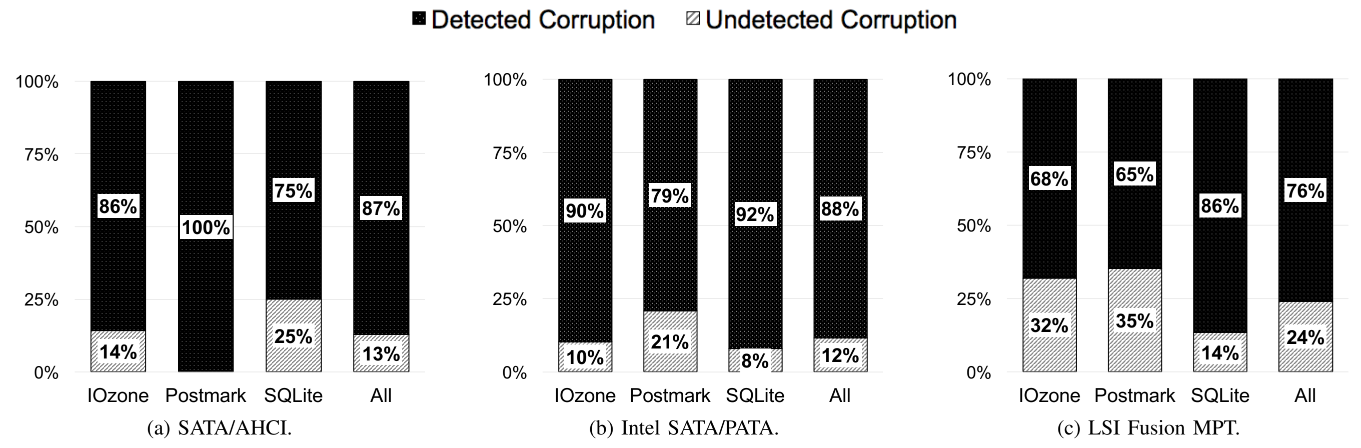


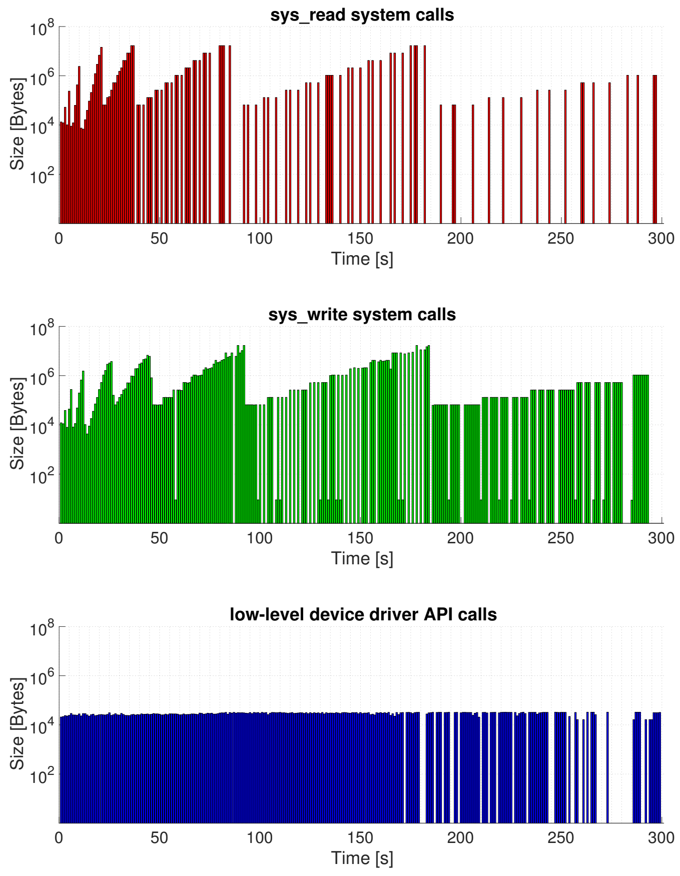
Fig. 11. Corruption detection coverage for the three target device drivers.

three workloads every 10 minutes. Moreover, we flush the I/O cache after each execution in order to do more stress on the I/O stack. We again check the correctness of the workloads and do not inject any fault during the experiment, in order to ensure that the execution is failure-free. We performed the experiment on each of the three virtualized storage interfaces (SATA/AHCI, Intel SATA/PATA, and LSI Fusion MPT from VMware ESXi). Moreover, we also performed the experiment on two additional configurations with two SATA/AHCI physical storage controllers (respectively, the Intel C600/X79 and the Intel 82801GR); both these physical controllers were installed on a bare-metal machine with an 8-core Intel Xeon 3.70GHz CPU and 16 GB RAM. The monitor is expected *not to detect* any failure, and DDP violations notified by the monitor are considered false alarms.

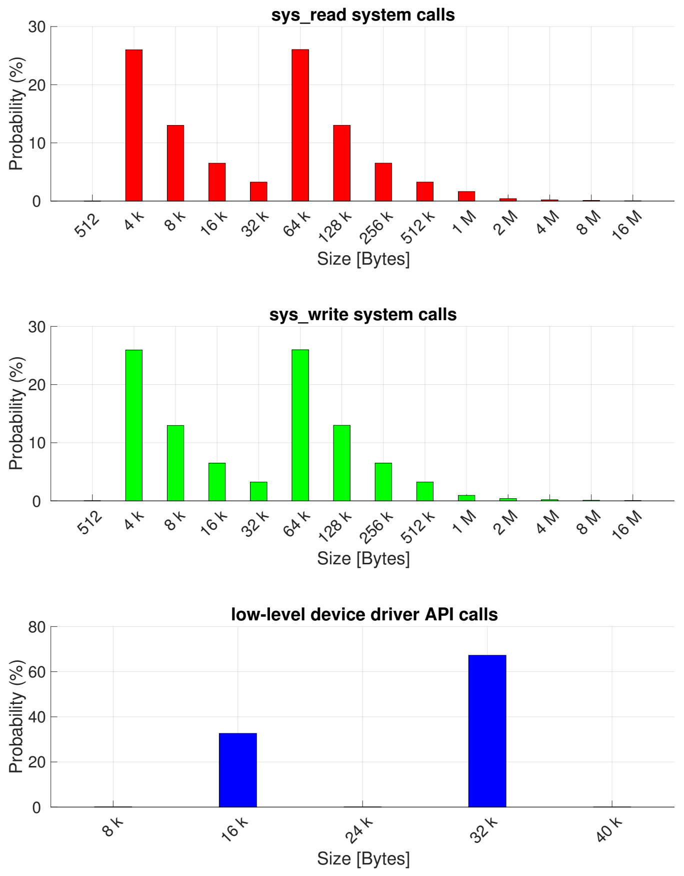
During these long-running experiments, the monitor did not raise any false alarm. This is an important aspect for the practical use of the proposed approach, as false positives hinder the adoption of monitoring techniques based on anomaly detection, due to the waste of efforts of system administrators. The rate of false positives that can be tolerated varies across users, as it depends on the cost of recovery actions and on availability requirements of the specific application (see also the discussion in § VI-A). The exposure of the monitoring approach to several cumulative weeks of stressful workloads provides some evidence that the approach can be deployed in practice without triggering false alarms for a long time.

Since no amount of stress testing can assure the total absence false positives, we must be careful before drawing conclusions. In general, any approach that uses dynamic program analysis is potentially exposed to false alarms, since training data cannot include all possible executions of the system. However, our filtering technique only extracts few, reliable parts of the controller interface, that are accurately modeled by the monitor. False positives are avoided by discarding the areas of the storage controller that exhibit frequent variations and do not contribute much to checking the I/O protocol. In this way, the monitor achieves a trade-off among coverage, false alarms, and performance overhead that is suitable for many practical scenarios.

We performed an additional experiment to better understand the reasons of this result, and to gain more confidence that the probability of false positives is small enough to be negligible for practical uses. Before this experiment, we hypothesized that, despite the high variability of the user workload (e.g., in terms of type and size of I/O system calls, process scheduling, etc.), the workload of the device driver at a low level (i.e., the I/O requests from the OS kernel to the device driver)



(a) I/O transfers over time.



(b) Distribution of I/O transfer size.

Fig. 12. An experiment to compare I/O access patterns at a high-level (system calls) and the corresponding low-level access patterns (device driver API).

follows simple, regular patterns, that are easy to learn by our monitoring approach even with a limited amount of training.

To get insights on this hypothesis, we executed the *IOzone* workload on the LSI Fusion MPT driver (the most complex of the three, as can be seen in Table I), and analyzed it from two perspectives (as showed in Fig. 12): the amount of data read and written by system calls of the application (the highest layer of the storage stack), and the amount of data transferred by individual I/O requests by the OS kernel to the device driver (a lower layer of the storage stack). Fig. 12b shows on the top how the read/written data vary over time, and on the bottom the corresponding I/O requests served by the device driver over time. For the same dataset, the Fig. 12b shows the distribution of the I/O transfer size respectively at the system call and at the device driver level. The analysis points out that even if the system calls at the application layer can significantly vary, the low-level I/O requests follow simpler patterns: when the volume of the system calls is low (first part of Fig. 12b), the low-level I/O requests are performed at a regular rate and with a fixed size; when the volume of the system calls increases, the low-level I/O requests exhibit more variability, but the most of individual I/O requests are limited to 16 or 32 fixed-size blocks per transfer (as pointed out in Fig. 12b). We attribute this behavior to the nature of the device driver API, which is designed to be simpler than the system call interface exposed to the users; moreover, the caching and buffering by the OS kernel also dampens variations. This behavior favors the monitoring of I/O storage activity at the device driver level, as this activity is easier to learn by the monitor than the activity at the user-space level.

E. Evaluation of performance overhead

We evaluated the overhead of the run-time monitoring approach, by comparing performance measures respectively *without* and *with* the monitor component. We again consider the IOzone, Postmark, and SQLite workloads, on the three storage device drivers. As in the previous section, we evaluate the approach both on three virtualized storage controllers (using the same setup presented in § V-A), and on two physical SATA/AHCI storage controllers with real HDDs.

We first analyze the impact of monitoring on the execution time of the workloads. Fig. 13 shows the average and the standard deviation of the execution time, that have been computed from 20 repeated executions of each workload. According to the tests in Fig. 13, monitoring only introduces a small overhead in most of the cases. The relative difference between execution times (lower is better) is only 0.5% in the best case (Intel SATA/PATA, *Postmark* workload). Moreover, the relative performance overhead (i.e., the gap between the

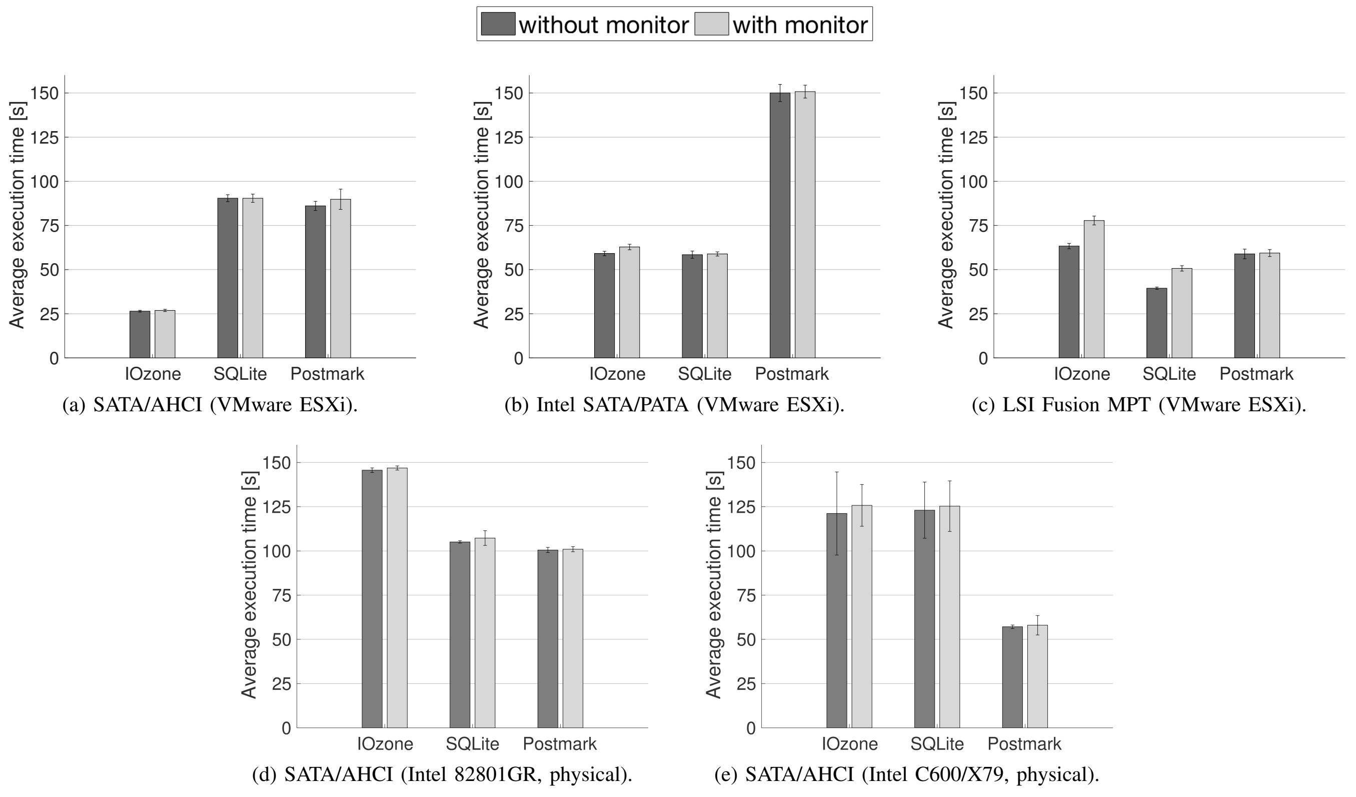


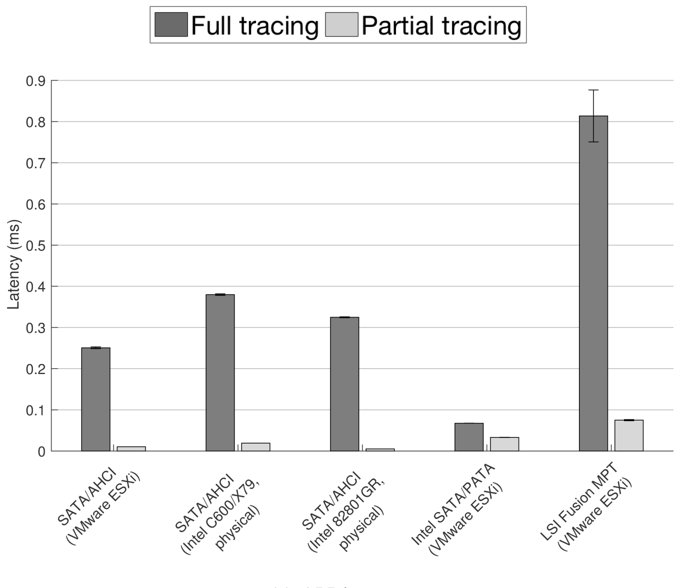
Fig. 13. Impact of the monitoring approach on the performance of the workloads.

executions with/without the monitor) for the physical SATA/AHCI controllers is comparable to the relative overhead that has been measured for the virtual SATA/AHCI controller. We found a noticeable difference only in the case of the LSI Fusion MPT device driver, which exhibited a relative difference of 27% in the worst case (*SQLite* workload).

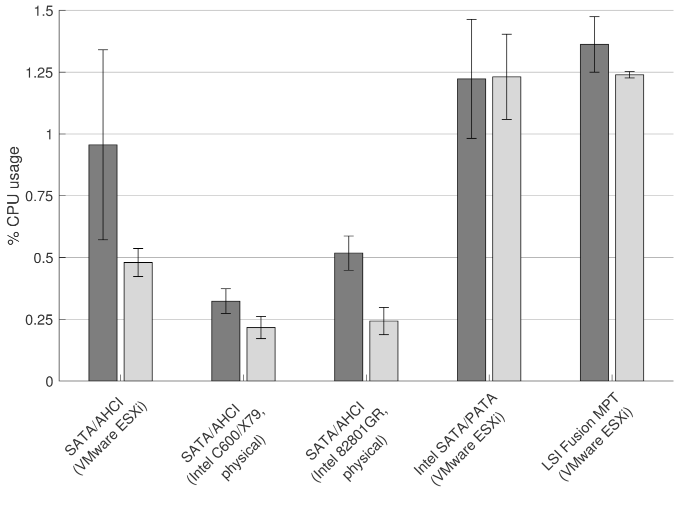
We attribute these differences to the overhead of dynamic probing, as every API call to the device driver triggers the breakpoint handler that is installed by our monitoring approach. In fact, the execution slow-down is only noticeable when the OS performs a higher number of driver API calls: for example, in the case of the *IOzone* workload, the OS performs about 90k API calls to the SATA/AHCI driver; under the same workload, the OS performs about 160k API calls to the LSI Fusion MPT driver. The number of API calls is determined by the nature of the API adopted by the device driver and the I/O subsystem: in the case of the LSI Fusion MPT, the kernel converts I/O system calls into a stream of smaller, and more numerous block transfers compared to the other drivers. In such cases, it is advisable (when possible) to configure the driver to increase the block transfer size, in order to reduce the number of API calls and the probing overhead; alternatively, to avoid dynamic probing by adopting less costly techniques to intercept API calls (e.g., by rewriting binary code before loading, as in more recent tracing techniques adopted in the Linux kernel [65], [66]), at the cost of making the approach more difficult to deploy in practice.

In another experiment, we evaluated the intrusiveness of the dynamic probing mechanism in more detail, by measuring the latency overhead imposed by the *prober* on the execution of driver APIs (in particular, to collecting data from the controller interface). Fig. 14a shows the time to execute our instrumentation code, without including the time to execute the driver API; therefore, these measures represent the *additional* latency that is introduced by the instrumentation (thus, in the case of no instrumentation, this measure would be zero). We evaluate this measures both when probing the full controller interface (i.e., the full tracing in the first step of § III-B), and when probing selected parts of the controller interface (i.e., the partial tracing in § III-B). We compare full and partial tracing to evaluate the importance of the trace filtering approach (§ III-B), since including all of the device interface in the model states worsens the overhead and accuracy beyond any practical use. Moreover, in Fig. 14b we show the CPU overhead due to work happening off the critical path (i.e., executed separately from the probed I/O operations performed by the driver), such as the collection of alarms from the kernel-level prober to a user-space logger. These activities off the critical path are executed by a user-space process provided by the SystemTap toolkit. Thus, we evaluate this performance overhead by measuring the CPU utilization of this process, which adds up to the normal CPU consumption of the workload and of the OS, and which is zero if we do not perform any instrumentation and collection. Again, we consider both full and partial tracing.

We notice that the gap between full and partial tracing is quite large with respect to the probing latency, with a slow-down of full tracing up to 24 times with respect to partial



(a) API latency.



(b) CPU utilization.

Fig. 14. Overhead of monitoring in terms of CPU utilization and additional API latency.

tracing (Fig. 14a), which is an unreasonable overhead for practical applications. Moreover, monitoring the full device interface leads to a high frequency of false positives (up to several false alarms per seconds), since the model includes noisy information that significantly varies between training and production runs (e.g., addresses and bit-fields). Thus, filtering is actually useful to reduce the overhead of probing, and to avoid the probe effect. Overall, the overhead under partial tracing (~1% CPU usage for off-loaded activities, and a probing latency of few tens of μs) can be considered low enough to be accepted for many practical scenarios.

VI. Discussion

In this section, we discuss about the deployment of the monitoring approach in practical scenarios, and about the problem of realistic fault injection.

A. Deployment of the proposed monitoring approach

We identify the administrators of the storage stack as one category of adopters of the proposed approach. A system administrator can enhance the reliability of the OS (e.g., device drivers from the Linux kernel such as the ones considered in our study) by introducing a monitoring module, and by leveraging the monitoring to trigger failure recovery strategies. From a practical point of view, the system administrator first needs to install the monitoring module, by identifying the DMA areas and memory-mapped registers of the device interface to be monitored, as discussed in section III-A, and to collect training traces. Then, the traces should be filtered using the automated techniques described in section III-B, in order to generate a new version of the monitoring module with the learned model, that will be deployed in operation.

Another use case for the proposed approach is that kernel developers release pre-built monitoring modules, and that others download and run the modules on their own system. In this case, the developer is responsible for providing an executable module that is already tailored to monitor the specific DMA areas and memory-mapped registers of the device interface. In this case, no kernel programming knowledge is required by the users. However, the user should still be responsible for executing a training workload, in order to let the pre-built module to learn a model for the storage controller, and make the monitor tailored for the actual configuration of the system. This training would not be feasible for developers since their environment may be different that the one of the user (e.g., because of a different configuration or chipset flavor of the storage controller). To evaluate the feasibility of this strategy, in this work we experimentally checked that the training can be performed with reasonable efforts, since the training does not require the user to apply complex workloads (e.g., it suffices to use popular and simple tools such as IOzone), and since the duration of the training is limited to few minutes.

Finally, another possible use case for the proposed approach is by system integrators (such as companies that commercialize "turn-key" solutions for data centers) to provide a pre-built computer server with additional high-reliability features to recover from faults in the storage stack. In this scenario, the system integrator has full control of the storage components and configuration, and can configure and pre-train the monitoring module for immediate use by its customers.

In order to decide how to react to an alarm raised by the monitoring approach, the user (e.g., a system administrator) should take into account the possibility that alarms are not due to faults from device drivers, but due to faults from the hardware devices, stressful conditions cased by the workload (such as low memory), and re-configurations of the storage (e.g., due to software upgrades, or to hardware maintenance). In principle, it is possible that external events (such as low memory and hardware faults), or even the execution a recovery procedure (for example, switching the driver to an alternative I/O mode, such as from interrupt-driven to polling or viceversa) may change the behavior of the device driver, thus causing violations with respect to the learned model. In the case that the system administrator has initiated an upgrade or maintenance action, he/she is already aware of the cause behind the violation, so the alarm from the monitoring tool can be recognized as harmless. If the device driver still raises alarms after the upgrade or reconfiguration, it can be necessary to re-train the model in order to fit the new configuration; this re-training is simple as it only requires few minutes and can be performed with a synthetic workload generator.

In other cases of alarm, it would be appropriate to still allow the driver to handle the operation, and to let the system administrator or a separate tool make decisions based on information from both the monitoring approach and from other sources (such as kernel logs and other diagnostic tools), in order to identify whether the alarm had been caused by an external event or by a failure of the device driver. In the case of monitoring alarms that happen together with other errors (such as error messages from the OS related to the memory allocator and the I/O management subsystem, etc.), the warning could be considered a consequence of low-memory or a hardware event. This decision can be made by looking for recent events that are explicitly logged by the device driver or the OS, or by querying the state of the device driver. Our monitoring approach is meant to be deployed side by side with the existing checks in the storage stack, to detect additional cases (i.e., silent protocol violations) that would otherwise not be detected by the kernel, and to support the diagnosis and handling of the problem by other tools or by administrators.

If an alarm from the monitoring approach raises suspicion of device driver failure, the end-user should temporarily stop the storage operations and perform thorough filesystem checks, which can identify the presence of data corruptions. The monitoring approach achieves a high-enough coverage and accuracy, thus avoiding to needlessly trigger the filesystem checks, and keeping the impact of checks low in terms of downtime and computational resources. Moreover, in the case of alarms, the end-user can undo and retry the recent I/O operations, or restore a previous version of the data. Having a timely warning from the monitoring approach helps to achieve a low time-to-detection, and thus to increase the likelihood of recovering the data in the case of failures of the storage driver. In any of these cases, the monitoring approach should be considered complementary to other error detection mechanisms in the I/O stack and should be accompanied by other tools and policies for handling the warnings according to the root cause of the problem.

B. Realistic injection of software faults

In this work, we have adopted a methodology based on fault injection to evaluate the monitoring approach. The aim has been to emulate software faults that do not surface during testing, and that lead to transient failures only when the driver is deployed in a production system. This behavior of software faults also allows us to train the monitoring approach with failure-free runs. However, it is important to note that it is difficult to reproduce the real software failure process in an experimental testbed, and that we need to approximate the failure process by accelerating its initial part, by forcing internal software errors, as defined by previous empirical studies on software mutations and fault injections.

Ideally, one would assess the failure detector with real failures, by running the target system for a long time, and by letting it to experience genuine failures caused by residual bugs in the device drivers. However, such real failures are relatively rare events: for example, two previous studies on field failure data on Windows 2K and NT machines (respectively 131 machines in the LAAS local area network [67] and 70 machines in a commercial organization [68]) reported a long time-between-failures of dozens of days on average, and up to several months for some machines; the time to failure is even higher when only considering disk driver failures. In our smaller testbed, OS failures are even more rare, and it would take years to observe a handful of corruption failures caused by storage drivers.

Another ideal solution would be to evaluate failure detection with real residual bugs, by carefully crafting inputs to trigger them. Unfortunately, this is not always possible, since device driver bugs are not triggered solely by inputs from applications, but also require subtle environmental conditions. For example, some bugs surface only when the disk controller is equipped with a specific chipset version (as disk device drivers typically have to support dozens of chipsets); but obtaining the right hardware for every bug would be unfeasible. In other cases, the driver should be stimulated by a specific interleaving of events (e.g., interrupts that overlap with the activity of asynchronous threads in the kernel or with a specific state of the memory allocator); but controlling these events would require a heavy instrumentation of the system (for example, by modifying the VM hypervisor to carefully controlling the timing of events for each bug). Moreover, this approach would bring an additional methodological issue: to have a large set of bugs, we would need to consider bugs across several versions of the device driver, so we would need to train and to evaluate the failure detector on each different version.

For these reasons, the typical approach to evaluate fault-tolerance solutions, such as previous papers on the reliability of device drivers, is to perform fault injection to accelerate failures. In general, (software) fault injection modifies either the code or the state of a running program, in order to force the occurrence of failures. In both cases, the program experiences failures that are not caused by the original code; this approach relies on the empirical observation that the injected mutations tend to cause failures that are similar to real ones. For example, Daran and Thévenod-Fosse [69] closely compared the variables corrupted respectively by real faults and mutations, and found a match in 85% of cases, even if the mutated program differs from the original program. Madeira et al. [70] and Christmansson and Chillarege [71] provided best practices for crafting mutations and errors to represent real failures. In the realm of mutation testing, empirical studies (such as the one from Andrews et al. [72]) have found that mutations are as difficult to detect as real faults. As a result, over the years, software fault injection has become an accepted approximation of real failures for research purposes.

In our study, we adopt a tool that emulates transient software failures of device drivers, by dynamically introducing faults at run-time in driver's code and state. The injections are *dynamic* in order to control the timing of failures, since it is still unfeasible (and still an open research problem, as we pointed out in a previous work [18], [19], [73], [74]) to automate a large number of experiments where the permanently-injected bugs (either injected pre-runtime, or real ones) are dormant during the training run and are triggered in a controlled way during the testing run. Therefore, during an experiment, the original (unmodified) device driver is first loaded in the OS; then, at run-time, the tool injects corruptions to emulate the transient effects of software faults, such as incorrect pointers and operands (e.g., NULL pointers that may circulate in the driver because of a sporadic failed memory allocation), wrong API calls (e.g., called prematurely or too late to be valid), incorrect control flows (e.g., an incorrect logical condition that does not handle an infrequent corner case).

It is important to note that all these effects may even be caused by code that is frequently executed, but they may only manifest themselves because of unlikely combinations of the timing of events, scheduling, hardware configuration, state of virtual memory, etc.. Since it is too cumbersome to reproduce the exact conditions that triggers the real bugs, software fault injection accelerates the failure process by forcing the faults' effects, in order to study how these effects lead to anomalies at the interfaces of the device driver (in our case, the hardware device interface). For these reasons, the fault injection tool used in this work is also adopted by many other recent studies on device driver reliability. Recent uses include research on Minix 3 fault-tolerant OS (using the HSFI tool [75], which is based on the same fault injector) and on quick rebooting techniques for the Linux kernel [76].

VII. Related work

The basic approach to tolerate *faulty device drivers* enhance the OS to provide *software fault isolation*. These techniques prevent a faulty driver from overwriting data or code outside its own, in order to avoid the escalation of the fault (e.g., even if a driver fails, the other functions of the OS are unharmed, and recovery actions can be initiated).

Software fault isolation relies on hardware and software solutions to avoid the propagation of errors from device drivers. The *Nooks* approach is one of the earliest and well-known [14]: it confines a driver into a *domain*, by dynamically updating the memory access permissions on the Memory Management Unit (MMU) before and after the driver's code is executed (i.e., by setting the pages of other kernel areas to read-only). Swift et al. [15] also proposed an approach to recover from failures of device drivers: once the failure is detected, a *shadow driver* (another copy of the driver that runs alongside the original one) becomes active and performs I/O requests that have been interrupted, in order to mask the failure from the user.

Other studies addressed faulty drivers by revising the architecture of the OS. The main approach has been to move device drivers from kernel-space (where the same memory address space is shared among all kernel components) into user-space processes (with their own memory address space). This idea has been applied in the *microdrivers* approach [77], and in *microkernel* OSes, such as Minix [16]. More recently the I/O Memory Management Unit (IOMMU) has been leveraged to protect the kernel from overwrites that may be caused by incorrect DMA transfers, which are otherwise not subjected to the checks of the MMU of the CPU [78], [79]. Finally, software-enforced fault isolation techniques have been developed to instrument drivers' code before loading them. One of the early techniques of this kind, by Wahbe *et al.* [80], inserts checks before every unsafe instruction (e.g., memory writes) in order to check destination addresses. SafeDrive [81] and BGI [82] guarantee type safety by placing checks at compile-time (such as assertions on pointers), which are enforced at run-time.

Software fault isolation segregates faulty device drivers, in order to prevent a driver from corrupting memory areas that belong to other kernel subsystems. However, these approaches are not meant to mitigate *corruptions* of memory that belongs to the faulty driver (e.g., commands and data exchanged with the I/O device). This limitation is especially problematic for storage device drivers, since drivers' failures may affect the users' data that are written on the persistent storage, and the corruptions may be undetected until it is too late to recover the data. Moreover, many of the studies on drivers' reliability were not focused on storage device drivers, and in many cases did not consider storage drivers at all due to their technical complexity (e.g., because of the large number of OS APIs involved and the large size of the device controller interface), focusing instead on simpler devices, such as network controllers and soundcards [15], [16].

In the field of storage reliability, many techniques and tools focus on filesystems, e.g., to improve error handling in filesystems with respect to errors raised by device drivers [83], [84], where data corruptions are healed with filesystem repair tools (e.g., *fsck*) [85]. However, these tools are limited to preserving the consistency of metadata (e.g., inodes and directories), but do not address corruptions of data blocks. Moreover, these tools are meant to be used sporadically (e.g., periodically, after a reboot, or after a major failure), and have a significant overhead [86].

The problem of protecting hardware devices from faulty device drivers has only been considered in very recent works. *Guardrail* [87] is an early solution, which uses an hypervisor to detect data races and memory access failures. Other approaches have introduced end-to-end consistency checks between the filesystem and the device driver (e.g., by computing and storing checksums of the data before passing them to the driver), and at the application level (e.g., by replicating blocks and comparing to each other) [84], [88]–[90]. However, the utility of such checks is limited since *data corruptions are only detected when the data is read again*, but a long time may elapse until a data block is read after it is corrupted, and these solutions incur in significant overheads.

To improve the reliability of storage device drivers, we present an approach that monitors at run-time the interactions between the device driver and the storage controller, in order to detect incorrect I/O operations as soon as they occur. Compared to our previous work [91], this paper presents a deeper experimental analysis to understand the reasons why the proposed approach achieves a good trade-off between coverage, accuracy, and overhead, including: a sensitivity analysis with respect to the duration of the training workload; a low-level analysis of the operational profile of the device drivers; an analysis of how the proposed approach complements the existing error detection mechanisms; a detailed analysis of the sources of overhead and how the approach reduces the data collection. Moreover, we experiment across three diverse and complex device drivers from the Linux kernel, covering different I/O standards and different Linux subsystems.

VIII. Conclusion

In this paper, we proposed an approach for detecting protocol bugs in storage device drivers at run-time, by monitoring the interfaces between the device driver and the other layers of the I/O storage stack, namely the storage controller and the OS kernel. The approach has been designed to ease its practical adoption: it avoids the need for source code and protocol specifications of the device driver, by automatically learning from execution traces; and it achieves a good trade-off between coverage, accuracy and overhead, by carefully selecting the parts of the device driver interface to be monitored.

We validated the proposed approach on three complex and popular device drivers from the Linux kernel. In particular, our experimental analysis has been aimed at understanding why the device driver protocols are amenable for run-time monitoring. One key finding is that the behavior at the device driver level, a lower layer of the I/O stack, is much simpler than the behavior in the uppermost layers of the stack, since the device driver performs smaller and less diverse types of I/O operations. Thanks to this property, it is sufficient to train the monitoring approach for few minutes in order to get a detector free from false positives, even if using a generic, synthetic workload generator. Another useful property is that we focus on selected parts of the storage interfaces (such as command codes and bitmasks) that are good and minimal indicators of protocol violation bugs, as the monitor has been able to detect most of the data corruptions caused by fault injection. Moreover, the monitor complements well existing error detection mechanisms (e.g., memory access checks by the CPU, and error logging in the Linux kernel), as these existing mechanisms cover the cases that are intentionally left out from our approach (e.g., incorrect memory addresses that cause memory access exceptions). Finally, we showed that the monitoring approach can be deployed with very little impact on the I/O performance, since we need to monitor only a small part of the storage interface, and we use dynamic probing mechanisms that are widespread among modern OSes and have a very small overhead.

Appendix

In this appendix, we provide technical information on the three device drivers that were analyzed in this study, and on the application of the proposed monitoring approach.

A. SATA/AHCI

Serial ATA (SATA) is a standard for a bus interface between storage controllers (*Host Bus Adapter*, HBA) and mass storage devices (including hard disks, optical disks, and solid-state drives), which leverages high-speed serial transmission [92]. The *Advanced Host Control Interface* (AHCI) is a popular open standard, proposed by Intel [93], for interfacing a SATA HBA to the OS kernel using a generic set of memory-mapped registers and in-memory data structures. AHCI and SATA are complementary standards (we will jointly refer to them as SATA/AHCI in the following), as they manage respectively the communication between the OS and the HBA, and between the HBA and the physical disk. Fig. 5a shows a simplified view of an SATA/AHCI storage stack architecture, from both the hardware and software perspectives.

In SATA/AHCI, the disk controller handles up to 32 ports. Furthermore, each port can manage up to 32 commands issued by the driver, which are written on the *command list*, an in-memory data structure. Commands and controls are enclosed in *Frame Information Structures* (FIS), along with flow control and error-detection codes to increase reliability. The device driver issues a command by writing a *command FIS*, which includes among others the command operation code, the logical disk block addresses to be read or written, etc.; subsequently, the disk controller fetches the FIS, and begins a data transfer; finally, on command completion, the disk controller raises an interrupt and writes on a different in-memory area, the *Received FIS Structure*, to notify the device driver about the status of the transfer (e.g., whether the command is completed without errors). The HBA exposes a set of *per-port* memory-mapped registers, which include:

- *Generic Host Control* (GHC): defines the capabilities and controls the behavior of the HBA (e.g., which ports the HBA exposes, and which capabilities the HBA supports, such as the *Native Command Queuing* (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. Intel SATA/PATA

The Intel SATA/PATA device driver in Linux supports storage controller chips in the PIIX series from Intel, including both serial (SATA) and legacy parallel (PATA) chipsets. In particular, we focus our evaluation on the Intel 82371AB/EB/MB PIIX4 PATA controller, as SATA controllers have already been covered by the SATA/AHCI device driver. The target PATA controller leverages the *Bus-Master DMA (BMDMA)* mechanism, a de-facto standard for ATA controllers, to allow the device to perform transfers in place of the CPU.

The device driver performs reads and writes by sending three command types: PIO, NO-DATA, or DMA. The driver communicates with the controller by writing information (denoted as *taskfile*) on a set of memory-mapped registers defined by the ATA standard, which include the command operation, the logical block addresses (LBAs) involved in the transfer, and so on. The IDE SATA/PATA driver provides two separate *IDE-channels* (with two groups of registers) for sending and receiving commands.

The physical memory region involved in a transfer to and from the hard disk is identified by a *Physical Region Descriptor* (PRD). PRDs are saved in a *Descriptor Table* in main memory. Each entry of that table is identified by an address, and by the size or transfer count of the region in bytes. As for the SATA/AHCI HBA, the Intel SATA/PATA HBA provides a set of memory-mapped registers through the *Bus Master Interface Base Address Register* (BMIBA), which is the base address for the Bus Master interface registers. These registers include:

- *Bus Master IDE Command Register (BMIC)*: enables/disables the bus master capability for the IDE controller and controls the direction for DMA transfers. The BMIC register also provides information that the driver uses to indicate DMA capability of the IDE device;
- *Bus Master IDE Status Register (BMIS)*: indicates the status about the IDE device and state of the transfer (e.g., the interrupt status);
- *Bus Master IDE Descriptor Table Pointer Register (BMIDTP)*: provides the base memory address of the Descriptor Table.

C. LSI Fusion MPT

This driver provides support for disk storage controllers from LSI based on the Fusion-MPT (Message Passing Technology) architecture [94]. That architecture provides an open, unified interface to manage SCSI, Fibre Channel, and Serial Attached SCSI (SAS) disks. Using the Fusion-MPT architecture, the host does not need to know the underlying bus protocol architecture to be able to communicate with the target devices.

The device driver and the disk controller communicate through two message queues, the *Request Message Queue* and the *Reply Message Queue*. Request messages trigger actions of the disk controller (e.g., write blocks on disk); reply messages contain status information about the disk controller. The elements in the queues are pointers to *message frames* that include a header, which uniquely identifies the message, and a payload, which contains information about the request (e.g., the SCSI CDB to specify the blocks to be accessed) or the reply (e.g., the error status of a completed command).

The addresses of the current request and reply message frames are pointed to respectively by the memory-mapped *Request FIFO Register* and the *Reply FIFO register*. The LSI Fusion MPT driver manages a memory pool for handling reply and request frames. When the driver issues a command, it asks for an MPT request frame from the pool, and creates a SCSI request mapped to a DMA area. A pointer to the message descriptor (i.e., the address that points to the requested frame) is then stored on the Request FIFO Register. Reply message queues are notified by the disk controller by setting the value of *Host Interrupt Status* register, and raising an interrupt to be handled by the device driver. We used our *prober* to perform a preliminary analysis of side effects of registers of this storage interface (as discussed in III-A). We found that reading from the *reply FIFO* register using our prober causes I/O failures, since the read has the side effect of clearing the interrupt register. Therefore, blacklisted this specific register, and probed all the remaining registers of the interface.

D. Applying the monitoring approach

We deployed the monitoring approach to the three targeted device drivers, by identifying the Linux kernel modules of these drivers (as showed in Fig. 5), and their API functions exposed to the upper layers of the Linux kernel.

The SATA/AHCI device driver is actually separated into two kernel modules: *ahci* and *libahci* (Fig. 5a). As soon as the *ahci* module is loaded, it requests and maps the memory regions needed for the disk controller, registers an interrupt handler, and resets the disk controller. On the other hand, the *libahci* module implements all the low-level mechanisms in order to communicate with the disk controller. Both the *ahci* and *libahci* kernel modules interact with the *libATA* [95] library, which provides a kernel interface for all ATA and SATA device drivers. All commands towards the disk controller are issued invoking the *ahci_qc_issue* function, which is implemented by the *libahci* module.

The Intel SATA/PATA device driver (Fig. 5b) consists of the *ata_piix* kernel module. When loaded, the *ata_piix* module registers the PIIX ATA PCI device using the kernel bus PCI API. The *ata_piix* leverages the *libATA* library for interacting with ATA devices. In particular, the *libata-sff* module implements the *ata_bmdma_qc_issue* API to send a taskfile towards a BMDMA controller.

Finally, the LSI Fusion MPT device driver is organized in three kernel modules: *mptbase*, *mptscsih*, and *mptspi* (Fig. 5c). The *mptbase* kernel module initializes the controller and provides all the basic functionalities of the MPT-Fusion protocol [94]. The *mptscsih* and the *mptspi* kernel modules are used specifically for MPT-Fusion devices that use the SCSI bus. In particular, the *mptspi* kernel module implements the *mptspi_qcmd* function that creates a SCSI request and sends it to the disk controller.

As soon as the identified APIs for issuing a command to the disk controller are invoked, we collect a snapshot of memory-mapped registers and in-memory data structures for the specific device driver. We compute the address of the memory-mapped registers using the bus APIs functions provided by the Linux kernel (e.g., *pcim_iomap_table*). The addresses of in-memory, DMA-mapped areas are obtained by probing the DMA APIs when loading the drivers (e.g., *dma_alloc_coherent*). In the case of the LSI Fusion MPT driver, we manually identified two arrays in DMA areas (the request and reply queues), and configured the monitor to separately handle the array elements. We implemented the monitor using *SystemTap* [24], by probing the API functions and inspecting the registers and in-memory data structures used by the device drivers.

Acknowledgment

This work has been supported by UniNA and Compagnia di San Paolo in the frame of "Programma STAR" (project "FIDASTE"). Furthermore, we are grateful to Dr. Francesco Fucci for his technical support with the fault injection tool and with a previous version of this manuscript.

References

[1] Oracle Corp., "Best practices for data reliability with Oracle VM Server for SPARC," 2010, http://www.oracle.com/technetwork/articles/systems-hardware-architecture/vmsrvrsparc-reliability-163931.pdf.

[2] IBM Corp., "Disaster recovery strategies with Tivoli Storage Management," 2002, http://www.redbooks.ibm.com/redbooks/pdfs/sg246844.pdf.

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

[4] A. Ganapathi, V. Ganapathi, and D. A. Patterson, "Windows XP kernel crash analysis," in *LISA'06*.

[5] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: Ten years later," in *ASPLOS'11*.

[6] 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.

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

[8] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with DDT," in *USENIX ATC'10*.

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

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

[11] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[12] G. Canfora, F. Mercaldo, and C. A. Visaggio, "An HMM and structural entropy based detector for Android malware: An empirical study," *Computers & Security*, vol. 61, pp. 1–18, 2016.

[13] L. Mariani, F. Pastore, and M. Pezze, "Dynamic analysis for diagnosing integration faults," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 486–508, 2011.

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

[15] 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.

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

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

[18] D. Cotroneo, R. Natella, and S. Russo, "Assessment and improvement of hang detection in the Linux operating system," in *SRDS'09*.

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

[20] "IOzone filesystem benchmark," http://www.iozone.org/, accessed: 2015-08-01.

[21] E. Nemeth, G. Snyder, S. Seebass, and T. Hein, *UNIX System Administration Handbook*. Pearson Education, 2000.

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

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

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

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

[26] G. Hunt and D. Brubacher, "Detours: Binary interception of Win32 functions," in *USENIX Windows NT Symp. '99*.

[27] A. Albinet, J. Arlat, and J.-C. Fabre, "Characterization of the impact of faulty drivers on the robustness of the Linux kernel," in *DSN'04*.

[28] M. Mendonca and N. Neves, "Robustness testing of the Windows DDK," in *DSN'07*.

[29] "The PCI utilities," http://mj.ucw.cz/sw/pciutils/, accessed: 2017-11-03.

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

[31] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *CCS'07*.

[32] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *NDSS'10*.

[33] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *NDSS'11*.

[34] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers: Where the Kernel Meets the Hardware*. O'Reilly Media, Inc., 2005.

[35] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of Linux file system evolution," *ACM Transactions on Storage*, vol. 10, no. 1, p. 3, 2014.

[36] M. Khalil, *Storage Design and Implementation in vSphere 6: A Technology Deep Dive*. VMware Press Technology, 2017.

[37] Intel Corporation, "AHCI supported chipsets," https://www.intel.com/content/www/us/en/support/articles/000005642/technologies.html, accessed: 2017-11-03.

[38] Broadcom Ltd., "SAS/SATA/NVMe host bus adapters," https://www.broadcom.com/products/storage/host-bus-adapters, accessed: 2017-11-03.

[39] 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*.

[40] I. Irrera, J. Durães, H. Madeira, and M. Vieira, "Assessing the impact of virtualization on the generation of failure prediction data," in *LADC'13*.

[41] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "No PAIN, no gain? the utility of parallel fault injections," in *ICSE'15*.

[42] 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.

[43] M. Hosken, *VMware Software-Defined Storage*. Sybex publisher, 2016.

[44] A. Mauro, P. Valsecchi, and K. Novak, *Mastering VMware vSphere 6.5*. Packt Publishing, 2017.

[45] VMware Inc., "VMware vSphere VMFS," Tech. Rep., 2017.

[46] 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.

[47] K. Kanoun, Y. Crouzet, A. Kalakech, A. Rugina, and P. Rumeau, "Benchmarking the dependability of Windows and Linux using PostMark^{TM} workloads," in *ISSRE'05*.

[48] "Phoronix test suite," http://www.phoronix-test-suite.com/, accessed: 2015-08-01.

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

[50] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability: A study of field failures in operating systems," in *FTCS'91*.

[51] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, 2006.

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

[53] D. Cotroneo, A. Lanzaro, and R. Natella, "Faultprog: Testing the accuracy of binary-level software fault injection," *IEEE Transactions on Dependable and Secure Computing*, 2016.

[54] L. Ryzhyk, "On the construction of reliable device drivers," Ph.D. dissertation, PhD Thesis, School of Computer Science and Engineering, University of New South Wales, 2010.

[55] LKML mailing list, "Rework AHCI LPM handling a little," https://lkml.org/lkml/2015/4/18/76, accessed: 2018-02-14.

[56] ——, "scsi: aacraid: Fix command send race condition," https://lkml.org/lkml/2017/11/21/897, accessed: 2018-05-10.

[57] LSI Logic, "MPT Fusion Linux OS driver release notes (mptlinux-4.00.13.04-1)," Tech. Rep., 2007.

[58] LKML mailing list, "Storage related regression in linux-next 20120824," https://lkml.org/lkml/2012/9/9/6, accessed: 2018-05-10.

[59] Ubuntu Linux Launchpad, "LSI Logic MPT driver mapping of scsi device busy to scsi host+device busy leads to read-only ext3 fs remounts on VMware ESX Server," https://bugs.launchpad.net/ubuntu/+source/linux-source-2.6.22/+bug/137585, accessed: 2018-02-14.

[60] LKML mailing list, "JMicron JM20337 USB-SATA data corruption bugfix," https://lkml.org/lkml/2008/7/22/631, accessed: 2018-05-10.

[61] Linux-scsi mailing list, “[PATCH] - fusion - mptfc bug fix’s to prevent deadlock situations,” https://marc.info/?l=linux-scsi&m=114600847100560, accessed: 2018-02-14.

[62] LKML mailing list, “scsi: qla2xxx: Get mutex lock before checking optrom_state,” https://lkml.org/lkml/2017/8/9/848, accessed: 2018-05-10.

[63] W. E. Cohen, “Tuning programs with OProfile,” *Wide Open Magazine*, vol. 1, pp. 53–62, 2004.

[64] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, “An empirical study of injected versus actual interface errors,” in *ISSTA’14*.

[65] B. Gregg, “Choosing a Linux Tracer,” 2015, http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html.

[66] J. Evans, “Linux tracing systems & how they fit together,” 2017, https://jvns.ca/blog/2017/07/05/linux-tracing-systems/.

[67] C. Simache, M. Kaâniche, and A. Saidane, “Event log based dependability analysis of Windows NT and 2K systems,” in *PRDC’02*.

[68] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, “Failure data analysis of a LAN of Windows NT based computers,” in *SRDS’99*.

[69] M. Daran and P. Thévenod-Fosse, “Software error analysis: A real case study involving real faults and mutations,” in *ISSTA’96*.

[70] H. Madeira, D. Costa, and M. Vieira, “On the emulation of software faults by software fault injection,” in *DSN’00*.

[71] J. Christmansson and R. Chillarege, “Generation of an error set that emulates software faults based on field data,” in *FTCS’96*.

[72] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *ICSE’05*.

[73] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, “Representativeness analysis of injected software faults in complex software,” in *DSN’10*.

[74] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, “On fault representativeness of software fault injection,” *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[75] E. van der Kouwe and A. S. Tanenbaum, “HSFI: Accurate fault injection scalable to large code bases,” in *DSN’16*.

[76] K. Yamakita, H. Yamada, and K. Kono, “Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery,” in *DSN’11*.

[77] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, “The design and implementation of microdrivers,” in *ASPLOS’08*.

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

[79] S. Boyd-Wickizer and N. Zeldovich, “Tolerating malicious device drivers in Linux,” in *USENIX ATC’10*.

[80] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *SOSP’93*.

[81] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, “SafeDrive: Safe and recoverable extensions using language-based techniques,” in *OSDI’06*.

[82] 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 *SOSP’09*.

[83] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Model-based failure analysis of journaling file systems,” in *DSN’05*.

[84] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Improving file system reliability with I/O shepherding,” SOSP’07.

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

[86] 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.

[87] O. Ruwase, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, “Guardrail: A high fidelity approach to protecting hardware devices from buggy drivers,” in *ASPLOS’14*.

[88] 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*.

[89] R. Alagappan, A. Ganesan, E. Lee, A. Albarghouthi, V. Chidambaram, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Protocol-aware recovery for consensus-based storage,” in *FAST’18*.

[90] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions,” in *FAST’17*.

[91] D. Cotroneo, L. De Simone, F. Fucci, and R. Natella, “MoIO: Run-time monitoring for I/O protocol violations in storage device drivers,” in *ISSRE’15*.

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

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

[94] LSI Corporation, “Fusion-MPT Device Management User Guide,” 2007, https://docs.broadcom.com/docs/12353292.

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

Domenico Cotroneo (Ph.D.) is associate professor at the Federico II University of Naples. His main interests include software fault injection, dependability assessment, and field-based measurements techniques. He has been member of the steering committee and general chair of the IEEE Intl. Symp. on Software Reliability Engineering (ISSRE), PC co-chair of the 46th Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN), and PC member for several other scientific conferences on dependable computing including SRDS, EDCC, PRDC, LADC, and SafeComp.

Luigi De Simone received his MSc degree with honors in Computer Engineering in 2013, and the PhD degree from the Federico II University of Naples, Italy, working on reliability evaluation of Network Function Virtualization infrastructures, within the Dependable Systems and Software Engineering Research Team (DESSERT) group. His research activity focuses on fault injection and dependability benchmarking of operating systems and cloud computing infrastructures. He received the “Best Student Presentation Award” from the ISSRE 2014 Conference, and the “Best Paper Award” from the NetSoft 2015 Conference.

Roberto Natella (Ph.D.) is assistant professor at the Federico II University of Naples, Italy, and co-founder of the Critiware s.r.l. spin-off company. His research interests include dependability benchmarking, software fault injection, and software aging and rejuvenation, and their application in operating systems and virtualization technologies. He has been involved in projects with Finmeccanica, CRITICAL Software, and Huawei Technologies. He contributed, as author and reviewer, to several journals and conferences on dependable computing and software engineering, and he has been in the steering committee of the workshop on software certification (*WoSoCer*) held with recent editions of the ISSRE conference.