Fine-Grained Fault Tolerance using Device Checkpoints

Asim Kadav, Matthew J. Renzelmann, Michael M. Swift
Computer Sciences Department, University of Wisconsin-Madison
{kadav, mjr, swift} @cs.wisc.edu

Abstract

Recovering faults in drivers is difficult compared to other code because their state is spread across both memory and a device. Existing driver fault-tolerance mechanisms either restart the driver and discard its state, which can break applications, or require an extensive logging mechanism to replay requests and recreate driver state. Even logging may be insufficient, though, if the semantics of requests are ambiguous. In addition, these systems either require large subsystems that must be kept up-to-date as the kernel changes, or require substantial rewriting of drivers.

We present a new driver fault-tolerance mechanism that provides fine-grained control over the code protected. Fine-Grained Fault Tolerance (FGFT) isolates driver code at the granularity of a single entry point. It executes driver code as a transaction, allowing roll back if the driver fails. We develop a novel checkpointing mechanism to save and restore device state using existing power-management code. Unlike past systems, FGFT can be incrementally deployed in a single driver without the need for a large kernel subsystem, but at the cost of small modifications to the driver.

In the evaluation, we show that FGFT can have almost zero runtime cost in many cases, and that checkpoint-based recovery can reduce the duration of a failure by 79% compared to restarting the driver. Finally, we show that applying FGFT to a driver requires little effort, and the majority of drivers in common classes already contain the power-management code needed for checkpoint/restore.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability

General Terms Design, Reliability

Keywords Device Drivers, Checkpoints

1. Introduction

In most commodity operating systems, third-party driver code executes in privileged mode. Faulty device drivers cause many reliability issues in these systems [8, 37]. Hence, there has been significant research to tolerate driver failures using programming-language and hardware-protection techniques [3, 6, 15, 16, 23, 26, 46]. These systems execute the entire driver as a single isolated component. However, much of this work focuses on *detecting* failures and *isolating* drivers from the rest of the system. Few of these systems address how to *restore* driver functionality beyond simply reloading the driver, which may leave applications non-functioning.

Most driver-reliability systems do not try to restore device state and instead completely restart failed drivers [17, 42, 48], effectively resetting device state to a known-good configuration. The state-of-the-art mechanism for restoring driver functionality, shadow drivers [41], logs state-changing operations at the driver/kernel interface. Following a failure, shadow drivers restart the driver and replay the log in order to restore internal driver and device state. This resets the driver and device to a state functionally equivalent to its pre-failure state. This approach, complete driver isolation and logging for recovery, poses four problems:

1. *Too hard:* Shadow drivers must be written for every class of driver and must be updated when the interface changes. This adds a large body of code to the kernel requiring constant maintenance, which is a high barrier to adoption. Other systems require substantially rewriting drivers, which is also a barrier.
2. *Not enough:* Shadow drivers must encode the semantics of the kernel/driver interface. However, many drivers have proprietary commands that cannot be captured by a shadow driver common to an entire class, leading to incomplete recovery. Recent work showed that up to 44% of drivers have non-class behavior [21].
3. *Too expensive:* Shadow drivers must interpose on and log *all* invocations of a driver. Continuous monitoring imposes a performance cost, particularly on high-performance devices such as SSDs and NICs even when the critical I/O path is bug-free.
4. *Too slow:* Restarting a driver, the first step of log replay, can be slow (multiple seconds) due to complex initialization code and therefore may not be useful in latency-sensitive environments.

A key source of these problems is that prior systems seek *completeness*: applying to *all* driver code at *all* times. While this reduces the per-driver cost, it pushes up both development and runtime costs.

We developed a new driver fault tolerance mechanism to address these shortcomings called *Fine-grained Fault Tolerance* (FGFT). Rather than isolating and recovering from the failure of an entire driver, FGFT executes a driver *entry point* as a transaction and uses software fault isolation to prevent corruption and detect failures. On entry to a driver, a stub copies parameters to the driver code. Only if the driver executes correctly are the results copied back. If the call faults, FGFT destroys the copy to roll back driver state and fails the call.

In order to restore device state modified by a driver before faulting, we developed a novel *device state checkpointing* mechanism that can capture the device state. The stub captures a checkpoint before invoking the driver, and restores the checkpoint on failure. This mechanism leverages existing power-management code present in most drivers, which greatly reduces the development cost of adopting FGFT.

FGFT shifts the cost of driver fault tolerance to the faulty code. While shadow drivers and whole-driver isolation require up-front code for any instance of a class of drivers, FGFT instead requires small changes to the driver itself to support isolation and implement checkpointing. Where past isolation mechanisms interpose on *all* driver code and reduce its performance uniformly, FGFT *only* imposes a cost on entry points selected for isolation. Thus, the cost of executing a single call with fault tolerance may be higher with FGFT than other systems, but when applied only to code off the critical path it has much *lower* overhead because the critical code is left unchanged. Thus, one possible use for FGFT is to apply it selectively to vulnerable code suspected or known to have bugs.

The contributions of our work are:

- We describe Fine-Grained Fault Tolerance, a system consisting of a static analysis and code generation tool that provides isolation by executing each driver request on a minimal copy of required driver state. Our system can be used to isolate specific requests and we show from a study of published bugs that fine-grained isolation is practical since bugs only affect 14% of all entry points in buggy drivers.
- We demonstrate a *novel* mechanism to create device checkpoints on a running system. In a study of six drivers, we show that taking a checkpoint is fast, averaging only 20 μs.
- We show how to use checkpoints and transactional execution of driver code to provide fast recovery and remove the permanent overhead of monitoring *all* requests.
- We show that the implementation effort of FGFT is small: we added 38 lines of code to the kernel to trap processor exceptions, and found that device checkpoint code can be constructed with little effort from power-management code present in 76% of drivers in common driver classes.

We begin with an overview of the FGFT design.

2. Design Overview

FGFT is a system to tolerate faults in drivers using a *pay-as-you-go* model based on checkpoints for recovery. This system protects code from faults at the granularity of a single thread executing a single entry point. FGFT recovers from any failures that occur during the function. This can greatly reduce the cost of isolating and tolerating faults because far less code is affected.

We list four goals of providing fine-grained fault tolerance:

1. *Class independent.* Isolation and recovery should be independent of the driver-kernel interface and should be able to recover driver actions from proprietary commands.
2. *Low infrastructure.* Little new code should be added to the kernel in support of FGFT.
3. *Pay-as-you-go.* FGFT should not have a fixed minimum overhead of isolation or monitoring driver behavior. Furthermore, programmer effort should only be required when fault tolerance is desired.
4. *Fast recovery.* FGFT should restore driver functions quickly after a failure without affecting other threads concurrently executing in the driver.

The first goal enables FGFT to apply to a broad range of drivers, and the second reduces the adoption cost for an operating system. Pay-as-you-go ensures that for high-performance drivers, tolerating faults in code off the critical path has little cost. Fast recovery enables its use in latency-sensitive environments.

The two major components of FGFT are an isolation mechanism to prevent a faulty driver from corrupting the OS and to detect failures, and a recovery mechanism to restore the driver to a functioning state after a failure. We begin a discussion of our fault model to motivate our design choices.

2.1 Fault Model

A driver entry point is a driver function invoked by the kernel or applications to access specific driver functionality. Each driver registers a set of functions as entry points, such as to initialize the device or transmit a packet. Driver entry points can be invoked by applications multiple times in arbitrary order. Hence, drivers should not make assumptions about the order or past history of these invocations. FGFT provides fault tolerance at the granularity of a single entry point into a driver. In contrast, past systems treat the entire driver as a component with internal state.

As the driver executes, the FGFT isolation mechanism enforces fine-grained memory safety. It ensures that the driver entry point is only allowed to access data passed to the driver and its stack; access to anything else will be treated as a fault. FGFT detects faults in driver entry points in three ways. First, FGFT detects memory failures (such as null pointer dereferences) and reading/writing unintended kernel and driver structures. Second, FGFT uses marshaling to copy data in and out of the driver. Type errors and malformed structures that cause the marshaling to fail will be detected, although errors with compatible types (such as treating an array of bytes as an array of longs), will not be. FGFT on its own does not provide any semantic checks to enforce driver invariants. Hence, driver faults must be detected within the entry point where they occur. Otherwise, failures that are triggered when one entry point improperly sets a flag that another read and faults cannot be tolerated. Third, FGFT catches processor exceptions such as NULL pointer exception, general protection fault, alignment fault, divide error (divide by zero), missing segments, and stack faults. It triggers recovery if an exception arises within an isolated driver entry point.

We design for an open-source environment, and therefore trust the compiler to produce code that correctly accesses the stack. We also assume that the driver is unable to hang or damage the device, although it may misconfigure the device.

A key benefit of FGFT is that by operating on specific entry points it can be selective about what code should be hardened against faults. We call the entry points to be isolated *suspect*. The suspect code can execute in isolation while the remainder of the driver executes in the kernel at full speed. Hence, FGFT is useful when specific driver code is known to have problems, such as just-patched code or code with known but un-patched vulnerabilities. We identify at least three cases where a fine-grained model is useful:

1. *Untested code*: Device drivers often contain untested code such as chipset-specific code or recovery code that can be invoked safely using FGFT.
2. *Statically found bugs*: Often static analysis tools identify hard to find/trigger driver bugs with substantial false positive rates. FGFT can be integrated with existing static analysis tools until a fix is issued, which often takes considerable time. This approach limits the overhead to just the buggy code, just when it contains known bugs.
3. *Runtime monitoring tools*: Runtime monitoring tools flag incoming requests based on their parameters, such as a specific `ioctl` command code, or are enabled at run time through module parameters [24] or security tools [32]. FGFT can dynamically decide whether to execute code in isolation or unsafely at full speed.

In our evaluation we analyze a list of bugs and find that they only affect 14% of all driver entry points. Hence, limiting the cost of fault tolerance to affected entry points can be useful. We now describe the two major components of FGFT: isolation and recovery.

2.2 Fine-Grained Isolation

FGFT provides isolation by forcing suspect code to operate on a *copy* of driver and kernel data. This ensures that anything the entry point does will not be seen by other threads or the kernel until it successfully completes, and allows quick recovery after a failure by deleting the copy. Thus, FGFT creates a clean copy of data needed for a driver entry point on every invocation, which consists of all data referenced by the entry point: parameters, global driver variables, and global kernel variables.

We use entry points as the granularity of isolation because it closely matches internal driver structure: they provide a natural boundary for returning errors after a fault, and drivers already synchronize concurrent invocation of entry points. If two driver threads cannot run concurrently in the driver, then driver synchronization ensures that one of them blocks until the other successfully completes. Thus, FGFT reuses existing synchronization mechanisms to ensure that when suspect code runs, no other threads are active in the driver. This ensures that any changes to device state will not be seen until the entry point completes successfully or it fails and recovery completes.

FGFT provides entry-point isolation with a copy-in/out model of driver and kernel state when suspicious entry-points are invoked. FGFT uses static analysis and code-generation to generate another kernel module that contains suspect entry points instrumented for memory safety. FGFT also generates communication code containing marshaling routines to copy driver and kernel state necessary for executing these entry points in isolation. Since the static analysis to marshal the data structures required by the isolated copy can be imprecise, FGFT requires a programmer to annotate ambiguous data types in the driver code. In order to provide even finer control over when to provide fault tolerance, FGFT automatically inserts taps, which are predicates that can decide at runtime whether to invoke the normal or fault-tolerant version of an entry point.

FGFT detects faults through run-time memory safety checks that detect access to unreachable addresses – memory not passed as a parameter or allocated by the entry point. Since, FGFT generates the code for copy-in/out, it is able to provide fine-grained memory safety (base and bound validation [28]). Furthermore, for failure detection, FGFT interposes on kernel trap handlers and detects if the faults originate from the suspicious entry-points and accordingly triggers recovery.

2.3 Checkpoint based Recovery

FGFT relies on checkpoints prior to a driver entry point for recovery. Unlike log-based recovery, which requires knowing how to replay requests, checkpoints can restore state independent of how a function modifies driver state. For example, a checkpoint of a device prior to an `ioctl` call allows its state to be recovered no matter what the call does.

Log-based recovery is also slow enough that the technique may not be useful in latency-sensitive environments. The primary delay comes from probing the device all over again which cold boots the device and performs the initialization steps as shown in Figure 1. For example, during initialization a network driver probes for the device, verifies EEPROM contents, tests the device, and registers the device with the kernel.

The checkpoint of the driver's state in memory is captured automatically through the copy-in/out model of invocation. Suspect code always executes on a copy of the driver state, so the original data is unmodified and need not be restored. The major challenge, though, is the *device state*, which may be modified unpredictably by the driver. We therefore require that drivers provide a facility for capturing and restoring device state. Prior to invoking suspect code, FGFT can take a checkpoint, and following a failure, it can restore the checkpoint.

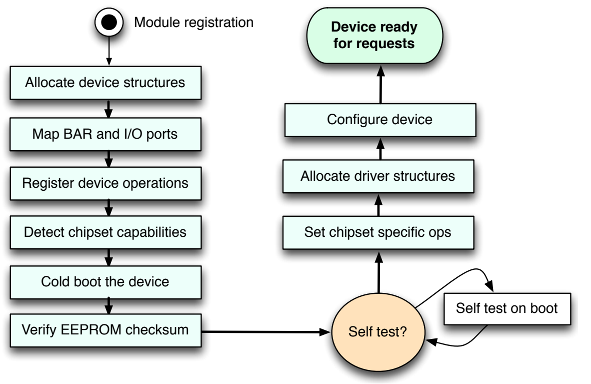


Figure 1. Modern devices perform many operations during initialization such as setting up kernel and device structures based on chipset and device features, checksumming device ROM data, various device tests followed by driver initialization and configuration.

An appealing approach is to treat devices like memory and copy memory-mapped I/O regions. However, reading registers may have side effects such as clearing counters. In addition, some devices overlay two logical registers, one for read and one for write, at the same address. Instead, we take inspiration from code *already present* in many drivers that must perform *nearly the same* task as checkpoint/restore: power management.

The functionality provided by power management, to suspend a device before entering a low power mode and restoring it when transitioning to high power mode is similar to what is required to support device checkpoints. We reuse the suspend/resume code by separating code that supports saving state to memory from the code that actually suspends the device. Similarly, we identify code required for restoring this state. In Section 4, we describe in detail how power management code can be re-factored to support checkpoint/restore in device drivers and how existing driver synchronization can be used to arbitrate device access.

2.4 Design Summary

FGFT improves the state of art in driver recovery and meets its goals. FGFT provides class-independent driver recovery with checkpoints as opposed to restarting the driver. Hence, FGFT discards failed requests and retains proprietary driver state such as *ioctls* that were issued before the failure.

FGFT requires very little kernel code, as the code for isolation is generated automatically and the recovery code requires only small modifications to existing driver code. The annotation cost for isolation and recovery is only required when a driver needs fault tolerance. Only when a suspicious request executes does FGFT execute it in isolation, thus limiting isolation overhead to these requests. Compared to FGFT, Nooks [42] and SUD [3] require a new kernel subsystem and writing and maintaining wrappers around the driver-kernel interface.

There is also no recovery overhead of monitoring correctly executing requests at *all* times since driver recovery is based on checkpoints. Finally, FGFT provides fast recovery since it does not restart the driver and re-execute the complicated device probe routines. The device state is restored from a checkpoint, so recovery is an order of magnitude faster as we demonstrate in our evaluation in Section 5.

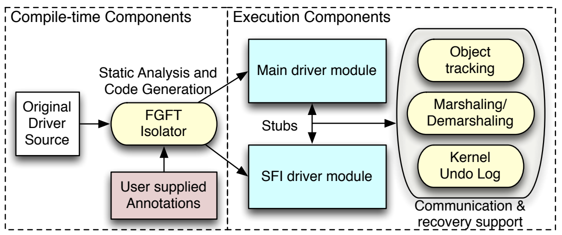


Figure 2. FGFT replicates driver entry points into a normal driver and an *SFI driver module*. A runtime support module provides communication and recovery support.

3. Fine-Grained Isolation

Isolation ensures that the driver and kernel state changes made by a request are not propagated if the request fails. We need the following properties from an isolation mechanism:

1. *Transactional execution*: We need to execute the driver entry points in a transactional fashion to keep a clean copy of all data modified by the driver.
2. *Memory safety and fault detection*: We need to ensure a driver cannot corrupt the kernel or other threads in the driver and provide mechanisms that detect when a driver has failed.
3. *Synchronization*: Threads executing in the driver need to synchronize with other threads to ensure they do not corrupt shared state in the kernel, driver, or device.

To achieve these goals, we rely on well-understood compile-time software fault isolation (SFI) [45]. As a driver entry point operates on data shared with the rest of the driver, the SFI mechanism must allow access to such data but prevent its corruption. FGFT therefore executes isolated code on a *minimal copy* of the driver and kernel, which is a copy of data referenced from an entry point but not entire structures. For example, when a network driver issues an ioctl to update its transmit ring parameters, FGFT uses points-to analysis and pre-determines the fields an entry point can access, such as `netdev→priv→tx_ring` and `netdev→priv→rx_ring`, and will only generate marshaling code to copy in/out only these fields to reduce the generated code and the unnecessary copying of unused fields. If the entry point does not fail, FGFT merges the copy back into the real driver and kernel structures. On a failure, the copy is discarded. In effect, FGFT executes the suspect entry point as a transaction using lazy version management [22].

However, not all data can be copied. Structures shared with the device, such as network transmit and receive rings, cannot be copied because the device will not share the copied structure. Instead, FGFT grants suspect code direct read and/or write access to these structures and relies on device-state checkpointing to restore these structures following a failure. Furthermore, driver code used for recovery cannot be isolated and must be trusted.

We implemented FGFT for the Linux 2.6.29 kernel. Figure 2 shows the components of FGFT. We describe how FGFT provides isolation, communicates with isolated code, and detects failures.

3.1 Software Fault Isolation

As FGFT targets open-source Linux device drivers, we implement SFI using a source-code rewriting tool called *FGFT Isolator* written using CIL [31]. It generates isolation code into the driver and produces communication code, described below, for communicating with the isolated code.

Isolator generates an additional driver module called the *SFI module* that contains a copy of all suspect entry points and all driver functions transitively called from those functions, instrumented for SFI. In addition, Isolator generates a new version of the driver that invokes the SFI module entry points. At the top of the existing entry points, Isolator inserts a test to see whether to execute normally or in isolation, and if so invokes the SFI module.

The decision to invoke a given entry point in isolation can be made in one of three ways. First, a developer can use the attribute `__attribute__((isolate))` to manually specify which functions to isolate. This causes the function to always execute with isolation. Second, FGFT can *automatically* use any static analysis tool to identify buggy code and which entry points are affected. These entry points are then always executed with isolation. Finally, the decision can be made at run time. A fault management system, such as the Solaris Fault management Daemon [39], can call into the SFI module and specify which functions to execute with isolation. Furthermore, it can register a function pointer at run time that takes the same arguments as the suspicious function and returns a decision of whether to isolate or not.

In addition to producing the SFI driver code, Isolator produces communication code that invokes the SFI driver and copies in the minimal driver and kernel state needed by the suspect entry points, copies out any changes made by the SFI driver, and initiates recovery following a detected failure. Isolator manages resource allocation, synchronization and I/O across the two copies. Isolator only detects memory failures. For other failures, such as arithmetic exceptions, we trap processor exceptions and check if they originate from SFI module.

Isolator uses CIL's memory tracking module [31] to instrument all memory references in the driver. It inserts a call to our `memcheck` function that verifies the target of a load/store is valid. If not, it detects a failure and invokes the recovery mechanism. The `memcheck` routine consults a *range table* to verify memory references and provide fine-grained memory protection by only allowing access to driver and kernel data as identified at compile time. This table contains the addresses and lengths of copied data structures and buffers shared with the device. The range table is created on every invocation of a suspect entry point and flushed on return.

We do not add all local variables to the range table because we trust the compiler to generate correct code for moving variables between registers and the stack. However, if the driver ever takes the address of a local variable, or creates an array as a local variable, then Isolator adds a call in the instrumented SFI driver to add the variable's address and length to the range table and remove it from the range table when the variable goes out of scope. Similarly, we trust the compiler to produce valid control transfers and do not instrument branch or call instructions.

3.1.1 Communication Code for Entry Points

FGFT Isolator generates *stub code* to invoke suspect entry points that copies data into and out of the driver. Similar to RPC stubs, these stubs create a copy of the parameters passed to the suspect code, but also copy any driver or kernel global variables it uses. When the suspect entry point completes, stub code copies modified data structures and return values back to the regular driver and kernel in the current thread. An alternative approach would be to rely on transactional-memory techniques to dynamically create a copy of data as it is referenced, which may have lower copying costs but higher run-time costs [2].

Isolator automatically identifies the minimal data needed for an entry point through static analysis. This data includes the structure fields from parameters referenced by the entry point or functions it calls plus fields of global variables referenced. As they copy data, stubs update the range table with the address and length of each object. For objects that cannot be copied (such as those shared with the device), stubs fill in the existing address of the field, its length, and whether the entry point needs read, write, or read/write access.

If suspect code invokes the kernel, Isolator generates stubs for kernel functions that copy parameters to the kernel and copies kernel return values back to suspect code. The SFI driver may pass in fields from its parameters to the kernel as arguments. To avoid creating a new copy of these fields, as would be done by RPC, FGFT maintains an *object tracker* that maps the address of kernel and regular driver objects to the address of objects in the SFI driver. Stub code consults the object tracker when calling into the kernel to determine whether arguments refer to a new or existing object. If an object exists, stubs copy the argument back to the existing object and otherwise temporarily allocate a new object. To support recovery, stubs may generate a *compensation entry* in a *kernel undo log*. This log records operations that must be reversed on failure, such as freeing memory allocations.

The stub code must know the layout of data structures and whether data is shared with the driver in order to correctly copy data. As driver code often contains ambiguous data structures such as `void *` pointers or list pointers (*e.g.*, `struct list_head`), we rely on programmer-provided annotations to disambiguate such code [48]. These annotations also declare which structure fields or parameters are shared with the device and should not be copied. In Section 5, we evaluate the difficulty of providing annotations.

Some driver functions trigger synchronous callbacks. For example, the `pci_register_driver` function causes a callback on the same thread to the driver's `probe` function. FGFT treats the callback as a *nested transaction*: it causes another isolated call operating on a second copy of the data.

3.1.2 Resource Access from SFI module

Some resources cannot be copied into the driver because they attach additional semantics or behavior to memory.

I/O memory. Driver entry points may communicate with the device by writing to I/O memory. Stubs grant the SFI driver read/write access to memory-mapped I/O regions and memory shared with the device via `dma_alloc_coherent`. Isolator identifies these regions with annotations and creates stubs that grant drivers direct read/write access.

Locks. Drivers synchronize with other threads using driver spin locks and mutexes. Hence, SFI grants read access to driver locks and calls the locking API to acquire/release the locks. The stub code for kernel locking routines add a compensation entry to the kernel undo log to release the lock after a failure. To ensure that changes made by suspect code are not seen by the rest of the kernel, the lock stubs defer releasing driver locks until after the entry point returns to the kernel. Apart from driver locks, drivers may acquire kernel-defined locks indirectly through kernel calls. FGFT does not expand the scope of these locks and releases them upon failure through compensation entries in the kernel undo log. Drivers can also directly manipulate kernel data structures while holding kernel-defined locks. FGFT will not recover correctly for such entry points. However, this is increasingly rare in Linux and there is an effort to ensure that kernel data structures are not directly exposed to drivers.

The above mechanism protects shared structures across driver threads. However, the suspicious thread can also block waiting for data to arrive on shared structures that have been copied over from other driver threads. Fortunately, resynchronization across driver threads is uncommon. Using static analysis, we measure how often driver threads wait for other threads using the Linux's `completion` family of functions or by polling in a loop.

Overall, we find driver resynchronization occurs in 2.7% of drivers and 1.4% of all entry points. Most re-synchronizations occur during communication with the device: drivers wait for a device operation to finish and a callback sets the completion structure. In most cases, only the completion structure responsible for device notifications needs to be annotated. However, complex drivers that communicate with devices using a layered interface, such as SCSI or WIFI, may wait for lower layers to communicate with device and update the appropriate drivers structures with the result of the operation. In such cases, annotations are required for completion structures and shared device structures for the driver to work correctly. Finally, driver threads also sleep inside loops waiting for other threads to finish by polling reference counts or driver structures. If these threads modify state across threads, then FGFT will not recover correctly for this fraction of drivers/entrypoints.

Memory allocation. Stubs for allocators invoke the kernel allocator, add the returned memory region to the range table and generate a compensation entry to free the memory on failure. The newly allocated memory is not copied into the driver because its contents do not need to be preserved. For kernel callbacks that implicitly allocate memory an appropriate compensation entry is generated.

3.2 Failure Detection

In addition to protecting the kernel and regular driver code from corruption, isolation provides the primary means to detect failures. FGFT's SFI mechanism implements *spatial memory safety* [28]: every memory reference must be within an object made accessible during the copy process. Thus, references outside the range table indicate a failure.

Stubs can detect additional failures when copying data back to the kernel. For example, if the driver writes an invalid address into a data structure, the copying code will dereference that address and generate an exception. We also modified the Linux kernel exception handlers to detect unexpected traps from the SFI driver as failures. If one occurs, the trap handler sets the instruction pointer to the recovery routine. This is the only change to the Linux kernel, and required 38 lines of code.

The detection mechanisms may miss several categories of failures. First, if the driver violates its own data structure invariants, stubs may not detect the problem. Recent work on identifying and verifying data structure invariants could detect these faults [5]. For example, if a suspect entry point sets a flag indicating that a field is valid but does not set the field, then corruption will leak out of the SFI driver. Second, the driver does not provide strong type safety, so the driver may assign a structure field to the wrong type of data. While this may be detected when the stub copies data, it is not guaranteed. Finally, FGFT does not enforce kernel restrictions on the range of scalar values, such as valid packet lengths.

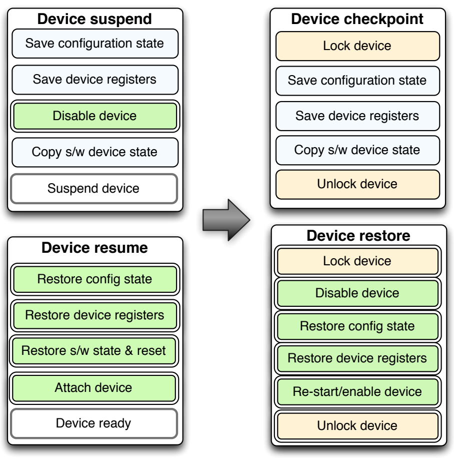
4. Checkpoint-based recovery

FGFT is built around checkpoint-based recovery. While checkpointing and restoring memory state is simple using techniques such as transactional memory or copy-on-write, it has not previously been possible to capture the state of a device. Without this, restoring memory state will lead to a driver that is inconsistent with respect to its device, believing incorrectly that it has performed an action or is operating in a different mode. We first describe device state checkpointing, which is the basis of FGFT's recovery mechanism. We then describe how FGFT uses device checkpoints to recover in case of a failure.

4.1 Device state checkpointing

To be useful, a device checkpoint mechanism should fulfill the following goals:

1. *Lightweight.* There should be no continuous monitoring or long-latency operations.



(a)

```
int rtl8139_checkpoint ( ... )

spin_lock_irqsave
      (&tp->lock, flags);

/* save PCI config state */
pci_save_state (pdev);

/* Copy out device state. */
dev->stats.rx_missed_errors +=
    RTL_R32 (RxMissed);


spin_unlock_irqrestore
       (&tp->lock, flags);
```

(b)

```
int rtl8139_restore (...)

spin_lock_irqsave
         (&tp->lock, flags);

/* disable device */
netif_device_detach (dev);

/* restore bus state */
pci_restore_state (pdev);

/* restart device */
if (!netif_running (dev))
    return 0;
rtl8139_init_ring (dev);
rtl8139_hw_start (dev);

RTL_W32 (RxMissed, 0);

/* re-enable device */
netif_device_attach (dev);
spin_unlock_irqrestore
         (&tp->lock, flags);
```

(c)

Figure 3. Our device state checkpointing mechanism refactors code from existing suspend-resume routines to create checkpoint and restore for drivers as shown in Figure (a). The checkpoint routine only stores a consistent device snapshot to memory while the restore loads the saved state and re-initializes the device. Figures (b) and (c) show checkpoint and restore routines in the rtl8139 driver.

2. *Broad.* The mechanism must work with a wide range of devices/drivers, including those with unique behavior.
3. *Consistent.* Drivers are often invoked on multiple threads, and checkpoints must be a consistent view of device state.

We identified the suspend/resume code already present in many drivers as having much of the functionality needed to implement checkpoint and restore. We next describe how power management for drivers works, and then describe how to reuse the functionality for driver recovery with checkpoints.

4.1.1 Suspend/Resume Background

Modern operating systems can dynamically reduce their power consumption to provide a hot *standby* mode, also called *suspend to RAM*, which disables processors and devices but leaves state in memory. One major component of reducing power is to disable devices. Thus, operating systems direct devices to switch to a low-power state when the system goes into standby mode. The behavior of devices is specified by the ACPI specification for the platform and by buses, such as PCI and USB.

In order to transition quickly between standby and full-power mode, drivers implement a power-management interface to save device state before entering standby mode, and to restore device state when leaving standby mode [12, 13]. These operations must be quick to allow fast transitions. The system-wide suspend-to-RAM mechanism saves the memory state of the driver, and the driver is responsible for saving and restoring any volatile device state. Drivers implement a power management interface with methods to save and restore state. For example, Linux PCI devices implement these two methods:

```
int (*suspend) (struct pci_dev *dev,
     pm_message_t state);
int (*resume) (struct pci_dev *dev);
```

When saving device state to memory, the driver may invoke the bus to save bus configuration data, as well as explicitly save the contents of select device registers that are not captured by the configuration state. The driver then instructs the device to suspend itself. Simple devices that have no state simply disable the device.

Upon resume, drivers wake the device, optionally perform a soft reset, restore their saved state. Since the latency of a system to respond post-resume is critical, the initialization is lightweight compared to restarting the driver, as it assumes the device has not changed. Similar to suspend, simple devices may just re-enable the device without restoring state.

For a system to support standby mode, all drivers must support power management. While not all drivers do (Linux is notorious for incomplete support [27]), it is widely implemented by Windows and MacOS drivers, and support in Linux drivers is improving.

The functionality provided by driver power management is very similar to what is needed for device state checkpointing. First, it provides the ability to save device state to memory in a way that allows applications to continue functioning. Second, even though the device may continue to receive power, the soft reset that occurs when re-enabling a device ensures that any previous state is replaced by the restored state. Finally, power management is implemented by most commonly used drivers. However, it is not directly usable for checkpointing: power management routines lack the ability to continue executing after suspending a device because the device has been disabled.

4.1.2 Checkpoint

Device state checkpointing is constructed from a subset of the device suspend support already present in drivers. A device may have many distinct forms of state, each of which require a different mechanism for checkpoint:

1. Device configuration information published through the bus configuration space.
2. Device registers with configuration data specific to the device.
3. Counters and statistics exported by the device and aggregated by the driver.
4. Addresses of memory buffers shared with the driver, such as the DMA ring buffers used by network drivers to send or receive packets.

We note that a checkpoint may not actually contain the full state of the device. Rather, it must contain *enough* information that functionality can later be restored without affecting applications. Thus, device state that can be recreated or recomputed need not be saved. Furthermore, the checkpoint only contains the device state. To be restored properly, it requires a consistent copy of the driver state taken at the same time. Thus, it must be paired with mechanisms such as transactional memory or copy-on-write to save the driver's state.

The configuration state is the easiest to save. Most buses provide a method to save configuration information. For example, PCI drivers in Linux use `pci_save_state`, that saves a set of standard registers and the base address registers (BARs) to memory. Each driver, though, must handle the remaining state, separately.

The driver explicitly saves register contents and counters in an internal driver structure. The difference between registers and counters arises during recovery, described below, because counter values cannot be written back to the device.

Memory buffers shared with the device can be recreated. As a result, most device drivers do not include the address of these buffers in a checkpoint. Instead, they free buffers during suspend and re-allocate them during resume.

Figure 3(a) diagrams the tasks performed by suspend and resume, and shows how that code is shuffled to create checkpoint and restore functionality. Of the suspend code, checkpointing reuses all the functionality except detaching the device with the kernel and suspending the device. As an example, Figure 3(b) shows the code to checkpoint the 8139too driver.

It may be necessary to checkpoint a driver while it is in use. Existing suspend routines assume the device is quiescent when the device state is saved. Checkpoint, though, may be called at any time. Thus, it must be synchronized with other threads using the driver. Because device state checkpointing must be coordinated with other mechanisms for capturing driver state, we do not put our own synchronization code in the checkpoint routine but reuse existing device locks in the driver. Device locks protect against conflicting configuration operations, or operations like resetting the device when I/O operations are in progress. This ensures that we do not corrupt device state assumed by another thread in progress when we reset device state in case of a failure.

4.1.3 Restore

The restore operation can be constructed from a mix of suspend and resume code. Normally the resume function is invoked when the device returns to full power and needs to be reconfigured. In the case of a checkpoint, though, the device is already running at full power. Thus, resume invokes the bottom half of the suspend routine to disable the device before restoring state.

The restore operation proceeds in four steps:

1. Disable the device to put it in a quiescent, known state.
2. Restore bus configuration state.
3. Re-enable the device.
4. Restore device-specific state.

Figure 3(c) shows the code to restore state for a simple network driver. Of the four categories of driver state, only configuration state and saved device registers can be reloaded. Counters, which cannot be written back to the device, are restored by adjusting the driver's version of the counter. Typically, the driver will read the device counter and update a copy in memory, and reset the device's counter. To restore the device's counter state, the driver only resets the device's counter; the in-memory copy of the counter must be saved as part of the driver's memory state.

To restore shared buffers, the driver releases existing shared buffers after disabling the device. As part of re-enabling the device, it recreates shared buffers and notifies the device of their new addresses. While this slows restore, it makes checkpoint very efficient because only irretrievable state is saved.

Unlike suspend-resume, it may be useful to use device state checkpointing from interrupt contexts where sleeping is not allowed. As a result, checkpoint and restore code must convert sleeps to busy waits (`udelay` in Linux) and use memory allocation flags safe for interrupt context (`GFP_ATOMIC` in Linux).

Fault Tolerance
Device recovery: Current recovery mechanisms require writing wrappers to track all device state and full device restart results in long latency.
OS functionality
Fast reboot: Restarting the OS requires probing all bus and device drivers.
NVM operating systems: Providing persistent state of a running system requires ability to checkpoint a running device.
I/O Virtualization
Device consolidation: Re-assignment of passthrough devices across different VMs needs to wait for device initialization.
Live migration: Live migration of pass-through devices converts the millisecond latency of migrations to multiple seconds due to device initialization.
Clone VMs: Ability to launch many cloned VMs very quickly is limited by device initialization.

Table 1. Other uses for fast device state checkpointing.

Compared to full-driver restart, resume improves performance because it does not re-invoke device probe, often the lengthiest part of starting a device normally. Furthermore, drivers for newer buses such as USB and IEEE1394 do not restart the device because the bus handles this operation. This further reduces restore times. For PCI devices, a further optimization is to avoid changing the power mode of the device. However, we observed that many drivers do not require actually powering down the device before performing restore. For these drivers, restore can be sped up by skipping these unnecessary power mode changes.

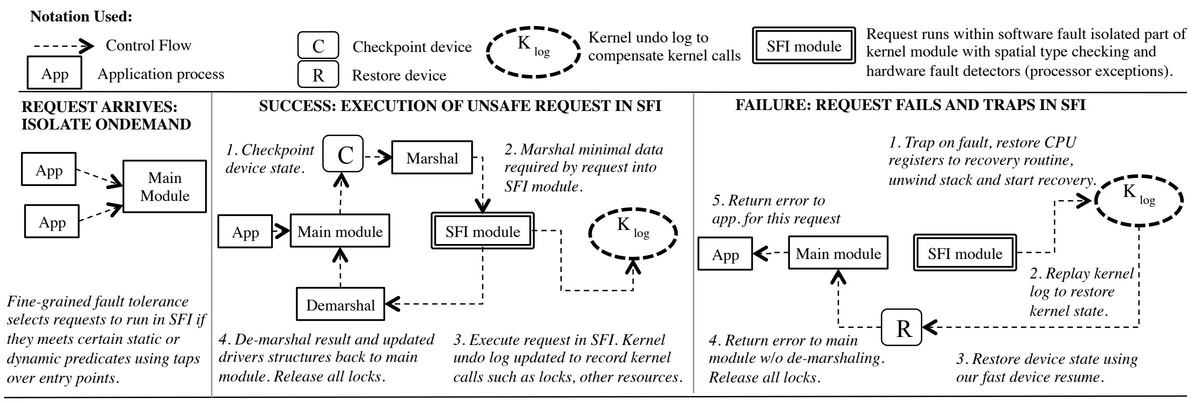
4.1.4 Discussion

Device state checkpointing provides several benefits compared with a logging approach to capturing driver/device state. First, it can be invoked at any time and has no cost until invoked. Thus, it has no overhead for infrequent uses. Second, it handles state unique to a device, such as configuration options. Correct standby operation demands that devices remain correctly configured across standby, and hence drivers must already save and restore any required state. However, device state checkpointing relies on power management code, which may not be present in all devices. It also requires a programmer to implement checkpoint/restore for every driver. We evaluate these concerns in Section 5.

Limitations. There is a risk in utilizing a mechanism for an unintended purpose: the driver continues running following a checkpoint and may thus further modify the device state. In contrast, devices are normally idle between suspend and resume. Thus, it is possible that the state saved is insufficient to fully restore the device to correct operation. However, the power management specifications require drivers to fully capture device state in software since devices can transition to an even lower power state where the device is powered off. In such cases, drivers must be able to be restore their original state, following a full reset. Thus, suspend must store enough information to restore from any state.

FGFT does not work with devices with persistent internal state, such as disks and other storage devices, since restore will only restore the transient device state and not the persistent state, such as the contents of files. As a result, use of checkpoints must be coordinated with higher-level recovery mechanisms, such as Membrane [40], to keep persistent data consistent.

Other uses for device state checkpointing. In addition to fault tolerance, device state checkpoints have other uses. Table 1 lists five possible uses. Within an operating system, checkpoints support fast reboot after upgrading system software by restoring device state from a checkpoint rather than reinitializing the driver.



Notation Used:
Control Flow
App
Application process
C
Checkpoint device
R
Restore device
K log
Kernel undo log to compensate kernel calls
SFI module
Request runs within software fault isolated part of kernel module with spatial type checking and hardware fault detectors (processor exceptions).
REQUEST ARRIVES: ISOLATE ONDEMAND
App
App
Main Module
Fine-grained fault tolerance selects requests to run in SFI if they meets certain static or dynamic predicates using taps over entry points.
SUCCESS: EXECUTION OF UNSAFE REQUEST IN SFI
1. Checkpoint device state.
C
Marshal
2. Marshal minimal data required by request into SFI module.
App
Main module
SFI module
K log
Demarshal
4. De-marshal result and updated drivers structures back to main module. Release all locks.
3. Execute request in SFI. Kernel undo log updated to record kernel calls such as locks, other resources.
FAILURE: REQUEST FAILS AND TRAPS IN SFI
1. Trap on fault, restore CPU registers to recovery routine, unwind stack and start recovery.
K log
5. Return error to app. for this request
App
Main module
SFI module
2. Replay kernel log to restore kernel state.
R
4. Return error to main module w/o de-marshaling. Release all locks.
3. Restore device state using our fast device resume.

Figure 4. **FGFT behavior during successful and failed entry point executions.**

Similarly, operating systems using non-volatile memory to survive power failures [1, 30] can restart drivers from a checkpoint rather than reinitializing the device. In virtualized settings, pass-through and virtualization-aware devices [33] allow drivers in guest operating systems to interact with physical hardware. Device state checkpoints enable virtual-machine checkpoints to a pass-through device [25] and live migration, because the device state from the source can be extracted and restored on identical hardware at the destination. With virtual devices, the latency of live migration can be as low as 60ms [10], so a 2 second delay to initialize a pass-through device adds significant downtime [20]. Finally, device state checkpointing enables dynamic fault tolerance at fine granularity as demonstrated by FGFT.

4.2 Recovery with checkpoints

We now describe how FGFT uses isolation and device checkpoints to perform recovery from failures. When a failure is detected, communication stubs call a recovery routine that is responsible for restoring correct driver operation.

Failure anticipation. To prepare for an eventual recovery, generated stubs create a device checkpoint before invoking a suspect entry point. They invoke the `checkpoint` routine. In addition, stubs for kernel functions log compensation entries to undo their effects in the kernel undo log. Driver state is not explicitly checkpointed; instead, suspect code operates on a copy of driver state as described in Section 3. In addition, the stub saves its processor register state, allowing a jump right into the stub if the driver fails.

Recovery steps. In case a failure is detected by SFI or processor exceptions originating from a suspect module, the recovery routine restores driver operation through a sequence of steps as shown in Figure 4:

1. *Unwind thread.* If not already in the stub, the instruction pointer is set to the address of the recovery code in the entry point's stub, which reloads the saved registers. Nested calls to drivers are logically handled as separate transactions, so there is no need to unwind the thread to the outermost entry point.
2. *Restore device state checkpoint.* The stub recovery code calls the driver's `restore` routine to restore the device state.
3. *Free call state.* All temporary structures created for the suspect entry point call such as the range table, object tracker, and copies of kernel/driver structures are released.
4. *Release locks.* Any locks acquired before or during the call to the SFI driver are released, allowing other threads to execute.

If a driver entry point fails, the stub returns an appropriate error indicator, such as a NULL pointer or an error code, and relies on higher-level code to handle the failure. As only the single entry point fails, this failure has little impact on applications. All application state relating to the device, such as open handles, remain valid. Furthermore, other threads in the driver continue to run as soon as the recovery process completes and releases all acquired locks.

Compared to other driver isolation systems, the recovery process is much simpler because only one thread is affected, so other threads are not unwound. In addition, the driver state is left unmodified, so it is not saved and restored. Finally, device state is restored quickly from a checkpoint rather than by replaying a log. Hence, we see that checkpointing device state results in quicker and simpler recovery semantics for driver recovery.

4.3 Implementation effort

A key goal for FGFT is to reduce the implementation effort to isolate a driver. FGFT consists of minimal modifications to the kernel exception handlers (38 lines of code), a kernel module containing the object tracker, range table, and recovery support, and the Isolator tool in OCaml. The module is 1,200 lines of C code, and Isolator is 9700 lines of OCaml that implement: SFI isolation (400 lines), stub generation (7,800 lines), and static analysis of references to parameter fields (1,500 lines). In comparison, Nooks adds 23,000 lines of *kernel code* (85x more than FGFT) to isolate and reload device drivers and shadow drivers add another 1,100 lines of code for recovery. FGFT also does not require any wrappers around the driver interface. Nooks required 14,000 lines of manually written wrappers, which are hard to maintain as the kernel interface changes. FGFT's isolator tool automatically generates similar code for stubs.

5. Evaluation

We implemented device state checkpoint and fine-grained fault tolerance for the Linux 2.6.29 kernel for six drivers across three buses. The evaluation examines the following aspects of FGFT:

1. *Fault Resilience.* What failures can fine-grained fault tolerance handle? We evaluate FGFT using a series of fault injection experiments and report our results.
2. *Performance.* What is the performance loss of fine-grained fault tolerance on steady-state operation? We report the performance cost for applying FGFT on support and core I/O routines.
3. *Recovery Time.* What is the downtime caused by a driver failure? We compare the time taken by FGFT to restore the device and cleanup the failed driver thread state with the time taken to unload and reload a driver.

Fault Type	Description of fault
Corrupt pointers	Dynamic: Corrupt all pointers referenced in a function to random values.
Corrupt stack	Dynamic: corrupt execution stack by copying large chunks of data over stack variable addresses.
Corrupt expressions	Static: corrupt arithmetic instructions by adding invalid operations (like divide by zero).
Skip assignment	Static: remove assignment operations in a function.
Skip parameters	Dynamic: zero incoming parameters in a function.

Table 2. Faults injected to test failure resilience represent runtime and programming errors. Dynamic faults are invoked using `ioctls`, and static faults are inserted with an additional compiler pass.

Driver	Injected Faults	Benign Faults	Native crashes	FGFT crashes
8139too	43	0	43	NONE
e1000	47	0	47	NONE
ens1371	36	0	36	NONE
pegasus	34	1	33	NONE
psmouse	22	1	21	NONE
r8169	46	0	46	NONE
Total	**258**	**2**	**256**	**NONE**

Table 3. Fault injection table with number of unique faults injected per driver. FGFT is able to correctly restore the driver and device state in each case.

4. *Usefulness of FGFT.* Is selectively isolating entry points useful? We evaluate whether suspect entry points can be identified in drivers and whether they reduce the amount of code isolated.

5. *Device Checkpoint Support.* Is re-using existing power management functionality reasonable? We examine the frequency of power management support in existing drivers that facilitates device checkpoints.

6. *Developer Effort.* What is the overhead to the developer to enforce isolation in the system? We measure the effort needed to annotate a driver for isolation and to add checkpointing code.

Unless otherwise specified, we compare FGFT against unmodified drivers running on an unmodified 2.6.29 Linux kernel.

5.1 Fault Resilience

We first evaluate how well FGFT can handle driver bugs using a combination of dynamic and static fault injection over six drivers. These tests evaluate both the ability of fine-grained fault tolerance to isolate and recover driver state as well as the ability of device state checkpointing to restore device functionality. Table 2 describes the types of faults inserted in the SFI module. Static fault injection modifies the driver source code to emulate programming bugs, while dynamic fault injection modifies driver data while running to emulate run time errors. We perform a sequence of trials that test each fault site separately.

During each experiment, we run applications that use the driver to detect whether a driver failure causes the application to fail. For network, we use `ssh`, and `netperf`, and for sound we use `aplay` and `arecord` from the ALSA suite. We tested the mouse by scrolling the device manually as we performed the fault injection experiments. After each injection experiment, we determine if there is an OS/driver crash or the application malfunctions. We re-invoke the failed entry point without the fault to ensure that it continues to work, and that resources such as locks have been released.

We injected a total of 258 unique faults in the native and FGFT drivers. Table 3 shows the number of faults injected for every driver and the outcome. For the native driver, all but two faults crashed the driver or resulted in kernel panics. The two benign faults were missing assignments.

In contrast, when we injected faults into driver entry points protected by FGFT, the driver and the OS remain available for every single fault. Furthermore, in every case, applications continue to execute correctly following the fault. For example, the sound application `aplay` skips for a few milliseconds during driver recovery but continued to play normally. The shell notes this disruption with the message "`ALSA buffer underrun`."

We also verify that internal driver and device state is correctly recovered using the `ethtool` interface for network drivers. We find that when failures happen while changing configuration settings, re-reading settings after a crash always returns correct values.

Finally, we verify that changes to drivers made using non-class interfaces, such as the `proc` and `sys` file systems, before injecting failures and present following recovery. In contrast, shadow drivers cannot replay these actions since they cannot capture non-class driver interactions.

5.2 Performance

The primary cost of using FGFT is the time spent copying data in and out of the SFI module and creating device checkpoints. We measure performance with a gigabit Ethernet driver, as it may send or receive receive more than 75,000 packets per second. Thus, the overhead of FGFT will show up more clearly than on low-bandwidth devices. We evaluate the runtime costs of using FGFT and regular versions of drivers in six configurations:

1. *Native:* Unmodified e1000 driver.
2. *FGFT $_{static}$:* Statically isolate 75% of code (all off I/O-path).
3. *FGFT $_{dyn-1/2}$:* Dynamically isolate every other packet in I/O path.
4. *FGFT $_{dyn-all}$:* Dynamically isolate every packet in I/O path.

The *dynamic* experiment measures the additional cost of choosing at runtime whether to invoke the regular or SFI version of a function. Finally, the *dyn-all* test represents the worst case of invoking the SFI module on the I/O path for a high-bandwidth device.

Our test machine consists of a 2.5 GHz Intel Core 2 Quad system congured with 4 GB DDR2 DRAM and an Intel 82541PI gigabit NIC running FGFT. We measure performance with netperf [18] by connecting our test machine to another machine with 1.2 GHz Intel Celeron processor and a Belkin NIC with a crossover cable. Table 4 reports the average of 5 runs.

In the *static* test where code off the I/O-path code is isolated, performance and CPU utilization are unaffected. These results demonstrate that FGFT achieves the goal of only imposing a cost on isolated code.

For the *dyn-1/2* test that isolates at runtime, entry points on the I/O path (the packet send routine) for every other packet, bandwidth dropped 4% and CPU utilization increased 0.5%. Thus, selectively applying isolation, even on critical I/O paths, has a small impact.

The performance drops further when we isolate critical path code on every request since we copy shared driver and kernel data across modules for *each* packet being transmitted. We find a 7.5% performance drop and 1% higher CPU utilization. FGFT is designed to limit isolation costs to specific requests and hence pays a cost of isolation because it needs to setup isolation (create copies) as each packet requests isolation. Overall, these results show that FGFT can be applied at no cost to high bandwidth devices off the I/O path and at marginal cost on the I/O path.

Device Locking: We run netperf using multiple threads to measure the overhead from device locks introduced at isolated entry points. We find that bandwidth drops an additional 2.6% and 4.6%

Single Threaded

System	Throughput	CPU
Native	843.6 Mb/s	2.5%
FGFT $_{static}$	843.6 Mb/s	2.5%
FGFT $_{dyn-1/2}$	811.5 Mb/s	2.9%
FGFT $_{dyn-all}$	784.4 Mb/s	3.4%

Multiple Threads

Threads	System	Throughput	CPU
10	Native	847.7 Mb/s	3.0%
	FGFT $_{dyn-all}$	791.4 Mb/s	4.2%
100	Native	941.7 Mb/s	4.2%
	FGFT $_{dyn-all}$	860.2 Mb/s	10.8%

Table 4. **TCP streaming send performance with netperf for regular and FGFT drivers with checkpoint based recovery.**

Driver	Class	Bus	Restart recovery	FGFT recovery	Speedup
8139too	net	PCI	0.31s	70μs	4400
e1000	net	PCI	1.80s	295ms	6
r8169	net	PCI	0.12s	40μs	3000
pegasus	net	USB	0.15s	5ms	30
ens1371	sound	PCI	1.03s	115ms	9
psmouse	input	serio	0.68s	410ms	1.65

Table 5. **FGFT and restart based recovery times. Restart-based recovery requires additional time to replay logs running over the lifetime of the driver. FGFT does not affect other threads in the driver.**

Driver	Class	Bus	Checkpoint times	Restore times
8139too	net	PCI	33μs	62μs
e1000	net	PCI	32μs	280ms
r8169	net	PCI	26μs	30μs
pegasus	net	USB	0μs	4ms
ens1371	sound	PCI	33μs	111ms
psmouse	input	serio	0μs	390ms

Table 6. **Latency for device state checkpoint/restore.**

with 10 and 100 threads respectively, as compared to FGFT running in a single thread. We also test the `ens1371` sound driver and find that playing multiple overlapping sound files does not result in any lags or distortions. These results show that FGFT introduces low overhead with high bandwidth devices running multiple threads.

5.3 Recovery Time

A major benefit of checkpoint-based recovery is the speed of restoring service. Table 5 lists the time taken by the driver to recover using FGFT and by unloading and reloading the driver. We measure recovery times by recording the time from detection of failure to completion of the restore routine. Overall, FGFT is between 1.6 and 4,400 times faster than restart recovery, and between 145ms and 1.5s faster. The drivers with the largest speedup have complicated probe routines that are avoided by restoring from a checkpoint. Hence, FGFT provides low-latency recovery and frequently offers an order-of-magnitude lower recovery latencies.

Checkpoint/restore latency. We examine the device checkpoint latencies to understand the source of our recovery performance in the previous section. Table 6 shows the latency of a checkpoint or restore for the same six drivers. Checkpointing is very fast, taking only 20μs on average and 33μs at worst. Thus, it is fast enough to be called frequently, such as before the invocation of most driver entry points. Restore times are longer, with a range from 30μs for the r8169 network driver to 390ms for psmouse. USB drivers store less state because the USB bus controller stores configuration information instead of the driver. Thus, during a normal resume, the bus restores configuration state before calling the driver to resume. The psmouse driver represents a legacy device and does not support

Class	Bus	Drivers reviewed	Drivers with PM	
net	PCI	104	68	65%
net	USB	32	27	84%
net	PCMCIA	4	4	100%
sound	PCI	72	63	88%
sound	USB	3	1	33%
sound	PCMCIA	2	2	100%
ATA	PCI	61	45	74%
SCSI	USB	1	1	100%
SCSI	PCMCIA	5	5	100%
Total	**-**	**284**	**216**	**76%**

Table 7. **Drivers with and without power management as measured with static analysis tools. USB drivers (audio and storage) support hundreds of devices with a common driver, and support suspend/resume.**

suspend/resume. Instead, we re-use existing driver code to reset the mouse.

5.4 Utility of Fine Granularity

We evaluate whether selectively isolating specific entry points is useful by looking for evidence that driver bugs are confined to one or a few entry points. If the functions with bugs are reachable through a large number of entry points, then full driver isolation is more useful than per-entry point isolation. For example, if a bug occurs in a low level read routine, then the bug will affect a large number of entry points.

In order to have a large data set, we use a published list of hardware dependence bugs [19] that represent one of the larger number of unfixed bugs in the drivers [11]. We were able to map these bugs in 210 drivers (541 total bugs) to our kernel under analysis. For each driver, we calculate the number of entry points and the fraction of code in the driver that must be isolated.

We find that the bugs are reachable through 643 entry points, for an average of 3 per driver. As a comparison, these drivers have a total of 4,460 entry points (21 per driver), so only 14% of entry points must be isolated. The code reachable from these entry points comprises only 18% of the code in these drivers. These results indicate that at least some classes of driver bugs are confined to a single entry point, and therefore FGFT can reduce the cost of fault tolerance as compared to isolating the entire driver.

5.5 Device Checkpoint Support

Device state checkpointing relies on existing power-management code. We measure how broadly it applies to Linux drivers by counting the number of drivers with power-management support. While modern ACPI-compliant systems require that all devices support power management, many legacy drivers do not.

We perform a simple static analysis over all network, sound, and storage drivers using the PCI, USB, and PCMCIA bus in Linux 2.6.37.6. The analysis scans driver entry points and identifies power management callbacks. Table 7 shows the number of drivers scanned by class and bus and the number that support power management. Overall, we found that 76% of the drivers support power management. Of the drivers that do not support power management, most were either very old, from before Linux supported power management, or worked with very simple devices. Only two modern devices, both Intel 10gb Ethernet cards, did not provide suspend/resume. Thus, we find that nearly all modern devices support power management and can therefore support device state checkpointing.

5.6 Developer effort

The primary development cost in using FGFT is adding annotations, which describe how to copy data between the kernel and the

Driver	Driver LoC	Isolation annotations	
		Driver Annotations	Kernel Annotations
8139too	1,904	15	20
e1000	13,973	32	
r8169	2,993	10	
pegasus	1,541	26	12
ens1371	2,110	23	66
psmouse	2,448	11	19

Table 8. Annotations required by FGFT isolation mechanisms for correct marshaling. Kernel annotations are common to a class, and driver annotations are specific to a single driver.

Driver	Recovery additions	
	LOC Moved	LOC Added
8139too	26	4
e1000	32	10
r8169	17	5
pegasus	22	4
ens1371	16	6
psmouse	19	6

Table 9. Developer effort for checkpoint/restore driver callbacks.

SFI module. Table 8 shows the number of annotations needed to apply FGFT to every function in each of the tested drivers. We separate annotations into *driver annotations*, which are made to the driver code, and *kernel-header annotations*, which are a one-time effort common to all drivers in the class. These annotations are the incremental cost of making a driver fault tolerant, and the implementation effort of Isolator and the kernel code described in Section 4 are the up-front cost.

Overall, drivers averaged 20 annotations, with more annotations for drivers with more complex data structures. Most driver classes required 20 or fewer kernel-header annotations except for sound drivers, which have a more complex interface and required 66 annotations. Thus, the effort to annotate a driver is only modest, as annotations touch only a small fraction of driver code. In comparison, SafeDrive [48] changed 260 lines of code in the e1000 driver for isolation and another 270 lines for recovery. Nooks [42] required 23,000 lines of code to isolate and reload drivers. Thus, these small annotations to drivers may be much simpler than adding a large new subsystem to the kernel.

Checkpointing implementation. We evaluated the ease of implementing device state checkpointing by adding support to the six drivers listed in Table 9. For each driver we show the amount of code we copied from suspend/resume to create checkpoint/restore as well as the number of new lines added. On average, we moved 22 lines of code and added six lines. The new code adds support for checkpoint/restore in interrupt contexts and avoids nested locks when the routines are invoked with a lock held. Even though device state checkpointing requires adding new code, the effort is mostly moving existing code. In comparison, implementing a shadow driver requires (i) building a model of driver behavior and (ii) writing a wrapper for every function in the driver/kernel interface to log state changes.

6. Related work

FGFT draws inspiration from past projects on driver reliability, program partitioning and software fault isolation systems.

Device driver recovery. Prior driver-recovery systems, including Nooks [42], Shadow drivers [41], SafeDrive [48] and Minix 3 [17] all unload and restart a failed driver. In contrast, FGFT takes a checkpoint prior to invoking the driver, and then rolls back to the most recent checkpoint, which is much faster. CuriOS provides transparent recovery and further ensures that client session state can be recovered [14]. However, CuriOS is a new operating system and requires specially written code to take advantage of its recovery system, while FGFT works with existing driver code in existing operating systems. ReViveI/O [29], and similar systems [35] provide whole-system checkpoint/restore by buffering I/O and only letting it reach the device after the next memory checkpoint. However, this approach does not work with polling, where I/O operations cannot be buffered and applied later.

Driver isolation systems. Driver isolation systems rely on hardware protection (Nooks [42] and Xen [15]) or strong in-situ detection mechanisms (BGI [6], LXFI [26], Mondrix [47] and XFI [43]) to detect failures in driver execution. However, in latter systems if the failure is detected *after* any state shared with the kernel has been modified then these systems cannot rollback to a last good state. Other driver isolation systems such as SUD [3] and Linux user-mode drivers [23], require writing class-specific wrappers in the kernel that are hard to maintain as the kernel evolves. FGFT differs from existing isolation systems by providing transactional semantics and limits the runtime overheads only to suspect code.

Software fault tolerance. Existing SFI techniques use programmer annotations (SafeDrive [48]) or API contracts (LXFI [26]) to provide type safety. XFI [43] transforms code binaries to provide inline software guards and stack protection. In contrast, FGFT operates on source code and allows drivers to operate on a copy of shared data. FGFT marshals the minimum required data and uses a range hash to provide spatial safety.

Transactional kernels. FGFT executes drivers as a transaction by buffering their state changes until they complete. VINO [38] similarly encapsulated extensions with SFI and used a compensation log to undo kernel changes. However, VINO applied to an entire extension and did not address recovering device state. In addition, it terminated faulty extensions, while most users want to continue using devices following a failure. FGFT is complementary to other transactional systems, such as TxOS [34], that provide transactional semantics for system calls. These techniques could be applied to driver calls into the kernel instead of using a kernel undo log of compensation records. Currently, these systems do not perform device I/O transactionally and either rely on higher-level atomicity techniques (TxOS [34] and xCalls [44]) or serialize transactions with a lock (TxLinux [36]).

Program Partitioning Program partitioning has been used for security [4, 7] and remote code execution [9]. Existing program partitioning tools statically partition user mode code or move driver code to user mode [16]. FGFT is the first system to partition programs within the kernel and is hence able to provide partitioning benefits to kernel specific components such as interrupt delivery and critical I/O path code. Furthermore, instead of partitioning code in any one domain, FGFT replicates its entry points and decides on a runtime basis whether a particular thread should run in isolation.

7. Conclusions

The performance and development costs of existing driver fault-tolerance mechanisms have restricted their adoption. In this paper, we presented fine-grained fault tolerance, a *pay-as-you-go* model for tolerating driver failures that can be dynamically invoked for specific requests. Fine-grained fault tolerance is made possible due to device checkpoints. This functionality is often considered to require a significant re-engineering of device drivers. However, we demonstrate that checkpoint functionality is already provided by existing suspend/resume code. While we only applied checkpoints to fault tolerance, there are more opportunities to use device checkpoint/restore, such as OS migration, fast reboot, and persistent operating systems that should be explored.

Acknowledgments

This work is supported by National Science Foundation (NSF) grants CNS 0745517 and CNS 0915363 and by a gift from Google. We would like to thank our shepherd Emmett Witchel for useful feedback. Swift has a financial interest in Microsoft Corp.

References

[1] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proc. of the 13th HOTOS*, 2011.

[2] A. Birgisson, U. E. Mohan Dhawan, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *Proc. of the 15th ACM CCS*, Oct. 2008.

[3] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In *USENIX ATC*, 2010.

[4] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proc. of the 13th USENIX Security Symposium*, 2004.

[5] S. Butt, V. Ganapathy, M. Swift, and C.-C. Chang. Protecting commodity OS kernels from vulnerable device drivers. In *Proc. of 25th ACSAC*, Dec. 2009.

[6] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proc. of the 22nd ACM SOSP*, 2009.

[7] S. Chong et. al. Secure web applications via automatic partitioning. In *Proc. of the 21st ACM SOSP*, 2007.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proc. of the 18th ACM SOSP*, 2001.

[9] B. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *Proc. of the 12th USENIX HotOS*. USENIX Association, 2009.

[10] C. Clark et. al. Live migration of virtual machines. In *Proc of the 2nd USENIX NSDI*, 2005.

[11] J. Corbet. Trusting the hardware too much. `http://lwn.net/Articles/479653/`. LWN February 2012.

[12] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Associates, Feb. 2005.

[13] M. Corp. Power management and ACPI - architecture and driver support. `msdn.microsoft.com/en-us/windows/hardware/gg463220`.

[14] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proc. of the 8th USENIX OSDI*, December 2008.

[15] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *OASIS Workhop*, 2004.

[16] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proc. of the 13th ACM ASPLOS*, Mar. 2008.

[17] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *Proc. of the 2007 IEEE DSN*, June 2007.

[18] R. Jones. Netperf: A network performance benchmark, version 2.1, 1995. Available at `http://www.netperf.org`.

[19] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Proc. of the 22nd ACM SOSP*, 2009.

[20] A. Kadav and M. M. Swift. Live migration of direct-access devices. *SIGOPS Operating Systems Review, 43:95–104*, 2009.

[21] A. Kadav and M. M. Swift. Understanding modern device drivers. In *Proc. of 17th ACM ASPLOS*, 2012.

[22] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[23] B. Leslie et. al. User-level device drivers: Achieved performance. *Jour. Comp. Sci. and Tech.*, 20(5), 2005.

[24] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc of the 26th ACM PLDI*, 2005.

[25] M. Mahalingam and R. Brunner. I/O Virtualization (IOV) For Dummies. `labs.vmware.com/download/80/`. VMworld 2007.

[26] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proc. of the 23rd ACM SOSP*, 2011.

[27] P. Mochel. The Linux power management summit. `http://lwn.net/Articles/181888/`, 2006.

[28] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *Proc. of the 30th ACM PLDI*, 2009.

[29] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: Efficient handling of i/o in highly-available rollback-recovery servers. In *Proc. of the 12th HPCA, 2006.*

[30] D. Narayanan and O. Hodson. Whole-system persistence. In *Proc. of the 17th ACM ASPLOS*, March 2012.

[31] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of the 11th CC*, 2002.

[32] V. Paxson. Bro: a system for detecting network intruders in real-time. In *Proc. of the 7th USENIX Security Symposium*, 1998.

[33] PCI-SIG. I/O virtualization. `http://www.pcisig.com/specifications/iov/`, 2007.

[34] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *Proc. of the 22nd ACM SOSP*, 2009.

[35] P. Ramachandran. *Detecting and Recovering from In-Core Hardware Faults Through Software Anomaly Treatment*. PhD thesis, University of Illinois, Urbana-Champaign, 2011.

[36] C. J. Rossbach et. al. TxLinux: Using and managing hardware transactional memory in an operating system. In *Proc. of the 21st ACM SOSP*, 2007.

[37] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proc. of the 4th ACM Eurosys*, Apr. 2009.

[38] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. *SIGOPS Operating Systems Review*, 30:213–228, 1996.

[39] Sun Microsystems. Opensolaris community: Fault management. `http://opensolaris.org/os/community/fm/`.

[40] S. Sundararaman et. al. Membrane: Operating system support for restartable file systems. In *Proc. of the 8th USENIX FAST*, 2010.

[41] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proc. of the 6th USENIX OSDI*, 2004.

[42] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. of the 19th ACM SOSP*, Oct. 2003.

[43] Úlfar Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: software guards for system address spaces. In *Proc. of the 7th USENIX OSDI*, 2006.

[44] H. Volos, A. Tack, N. Goyal, M. Swift, and A. Welc. xcalls: safe i/o in memory transactions. In *Proc of the 4th ACM Eurosys*. ACM, 2009.

[45] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM SOSP*, Dec. 1993.

[46] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proc. of the 8th USENIX OSDI*, 2008.

[47] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proc. of the 20th ACM SOSP*, 2005.

[48] F. Zhou et. al. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proc. of the 7th USENIX OSDI*, 2006.